SC38-0497-06

IBM Visualization Data Explorer

Programmer’s Reference

Version 3 Release 1 Modification 4

IBM

SC38-0497-06

IBM Visualization Data Explorer

Programmer’s Reference

Version 3 Release 1 Modification 4

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page xvii.

Seventh Edition (May 1997)

This edition applies to IBM Visualization Data Explorer Version 3.1.4, to IBM Visualization Data Explorer SMP Version 3.1.4, and to all subsequent releases and modifications thereof until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product. Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

A form for readers’ comments appears at the back of this publication. If the form has been removed, address your comments to:

IBM Corporation
Thomas J. Watson Research Center/Hawthorne
Data Explorer Development
P.O. Box 704
Yorktown Heights, NY 10598-0704
USA

If you send information to IBM, you grant IBM a nonexclusive right to use or distribute that information, in any way it believes appropriate, without incurring any obligation to you.

Contents

Figures

Tables

Notices

Products, Programs, and Services

Trademarks and Service Marks

AViiON	Data General Corporation
DEC	Digital Equipment Corporation
DGC	Data General Corporation
Graphics Interchange Format (GIF)	CompuServe, Inc.
Hewlett-Packard	Hewlett-Packard Company
HP	Hewlett-Packard Company
iFOR/LS	Apollo Computer, Inc.
Motif	Open Software Foundation
NetLS	Apollo Computer, Inc.
Network Licensing Software	Apollo Computer, Inc.
OpenWindows	Sun Microsystems, Inc.
OSF	Open Software Foundation, Inc.
PostScript	Adobe Systems, Inc.
X Window System	Massachusetts Institute of Technology

Copyright notices

IBM Visualization Data Explorer contains software copyrighted as follows:

THIS AGREEMENT REPRESENTS THE ENTIRE AGREEMENT CONCERNING THE PROGRAM AND DOCUMENTATION POSAL, REPRESENTATION, OR UNDERSTANDING BETWEEN THE PARTIES WITH RESPECT TO ITS SUBJECT MATTER. BY BREAKING THE SEAL ON THE TAPE, YOU ARE ACCEPTING AND AGREEING TO THE TERMS OF THIS AGREEMENT. IF YOU ARE NOT WILLING TO BE BOUND NY THE TERMS OF THIS AGREEMENT, YOU SHOULD PROMPTLY RETURN THE CONTENTS, WITH THE TAPE SEAL UNBROKEN; YOUR MONEY WILL BE REFUNDED.

1. License: ISC remains the exclusive owner of the Program and the Documentation. ICS grant to Customer a nonexclusive, nontransferable (except as provided herein) license to use, modify, have modified, and prepare and have prepared derivative works of the Program as necessary to use it.

2. Customer Rights: Customer may use, modify and have modified and prepare and have prepared derivative works of the Program in object code form as is necessary to use the Program. Customer may make copies of the Program up to the number authorized by ICS in writing, in advance. There shall be no fee for Statically linked copies of the Motif libraries. Statically linked copies are object code copies integrated within a single application program and executable only with that single application. Run Time copies require payment of ICS' then applicable fee. Run Time copies are copies which include any portion of a linkable object file (“.o” file), library file (“.a” file), the window manager (mwm manager), the U.I.L. compiler, a shared library, or any tool or mechanism that enables generation of any portion of such components; other copies will require payment of ICS' applicable fees. TRANSFERS TO THIRD PARTIES OF COPIES OF THE LICENSED PROGRAMS, OR OF APPLICATIONS PROGRAMS INCORPORATING THE PROGRAM (OR ANY PORTION THEREOF), REQUIRE ICS' RESELLER AGREEMENT. Customer may not lease or lend the Program to any party. Customer shall not attempt to reverse engineer, disassemble or decompile the program.

3. Limited Warranty: (a) ICS warrants that for thirty (30) days from the delivery to Customer, each copy of the Program, when installed and used in accordance with the Documentation, will conform in all material respects to the description of the Program's operations in the Documentation. (b) Customer's exclusive remedy and ICS' sole liability under this warranty shall be for ICS to attempt, through reasonable efforts, to correct any material failure of the Program to perform as warranted, if such failure is reported to ICS within the warranty period and Customer, at ICS' request, provides ICS with sufficient information (which may include access to Customer's computer system for use of Customer's copies of the Program by ICS personnel) to reproduce the defect in question; provided, that if ICS is unable to correct any such failure within a reasonable time, ICS may, at its sole option, refund to the Customer the license fee paid for the Product. (c) ICS need not treat minor discrepancies in the Documentation as errors in the Program, and may instead furnish correction to the Program. (d) ICS does not warrant that the operation of the Program will be uninterrupted or error-free, or that all errors will be corrected. (e) THE FOREGOING WARRANTY IS IN LIEU OF, AND ICS DISCLAIMS, ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY AND FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL ICS BE LIABLE FOR ANY INCIDENTAL OR CONSEQUENTIAL DAMAGES, INCLUDING WITHOUT

About This Book

This reference is intended for programmers who:

1. want to write their own modules for use with Data Explorer, or
2. want to write applications which incorporate Data Explorer modules or use the Data Explorer data model, or
3. want to write applications which directly control the Data Explorer executive or user interface. Programmers using this reference should be familiar with Data Explorer (in particular, its data model).

In addition to covering various aspects of creating and implementing modules, this reference describes in detail the use of the Module Builder—a utility that simplifies these tasks considerably.

Summary of Topics

- Chapter 1, "Overview" on page 1 briefly discusses the various ways the Data Explorer libraries can be used and points you to the appropriate sections, depending on what task you want to accomplish and how you want to use Data Explorer.
- Chapter 2, "Writing a Simple Module" on page 5 presents a simple example to outline the basic procedure for creating and implementing a module. It also summarizes the data and execution models of Data Explorer.
- Chapter 3, "Module Builder" on page 17, details the basic features of the Module Builder user interface.
- Chapter 4, "Working with Data" on page 31 shows how to write modules which are concerned only with "data" component of an object. A simple module which adds two fields together is described.
- Chapter 5, "Working with Positions" on page 37 shows how to write modules which operate on the "positions" component of an object. A simple "glyph" type module which places a mark at each position in a field is described.
- Chapter 6, "Working with Connections" on page 43 shows how to write modules which operate using the "connections" component of an object. A simple module which averages data over the nearby neighbors in a field is described.
- Chapter 7, "Importing Data" on page 47 shows how to write an import filter.
- Chapter 8, "Using the Pick Structure" on page 55 shows how to write a module which uses the structure output by the Pick tool to perform specific operations on objects "picked" using the mouse in the Image window.
- Chapter 9, "Writing Modules for a Parallel Environment" on page 67, explains how to write a module for execution on parallel processors.
- Chapter 10, "Making a Module Work" on page 79, deals with the main aspects of implementing a new module: module description files, compilation and linking, and debugging.
- Chapter 11, "Working with Data Model Objects" on page 95, details the programming interface of the Data Explorer data model.
- Chapter 12, "System Services" on page 113, Chapter 13, "Data Processing" on page 131, and Chapter 14, "Geometric Objects" on page 145 summarize the Data Explorer routines available for:
  - System services (e.g, error handling and storage allocation)
  - Data processing (e.g, partitioning and hashing)
  - Creating geometric objects.
- Chapter 15, "Rendering" on page 149, deals with several advanced aspects of rendering an image: transformations, shading, and tiling.

- Chapter 16, “DXLink Developer's Toolkit” on page 157, describes the main features of this aid to application programming for Data Explorer.
- Appendix B, “Data Explorer Data Model Library: DXlite Routines” on page 181, lists the subset of Data Explorer routines for creating, querying, and modifying Data Explorer Objects.
- Appendix C, “Data Explorer Library Routines” on page 183, lists and describes all the Data Explorer interface routines.
- “Glossary” on page 375, a glossary of Data Explorer terms, follows the appendices.

Typographic Conventions

Boldface	Identifies commands, keywords, files, directories, messages from the system, and other items whose names are defined by the system.
Italic	Identifies parameters with names or values to be supplied by the user.
`Monospace`	Identifies examples of specific data values and text similar to what you might see displayed or might type at a keyboard or that you might write in a program.

Related Publications and Sources

IBM Publications

- *IBM Visualization Data Explorer User’s Guide*, SC38-0496

  Details the main features of Data Explorer, including the data model, data import, the user interface, the Image window, and the visual program editor. and the scripting language. Of particular interest to programmers: chapters on the data model and the scripting language.

- *IBM Visualization Data Explorer User’s Reference*, SC38-0486

  Contains detailed descriptions of Data Explorer’s tools.

  Note: Consult this reference if you are creating visual programs or scripts.

- *IBM Visualization Data Explorer Programmer’s Reference*, SC38-0497

  Contains detailed descriptions of the Data Explorer library routines.

  Note: Consult this reference if you are writing your own modules for Data Explorer.

Non-IBM Publications

The following treat various aspects of computer graphics and visualization:

Adobe Systems Incorporated, *PostScript Language Reference Manual*, 2nd Ed., Addison-Wesley Publishing Company, Massachusetts, 1990.

Aldus Corporation and Microsoft Corporation, *Tag Image File Format Specification, Revision 5.0*, Aldus Corporation, Washington, 1988.

Arvo, Jim, ed., *Graphics Gems II*, Academic Press, Inc., Boston, Massachusetts, 1991.

Foley, J.D., van Dam, A., Feiner, S.K., Hughes, J.F., *Computer Graphics: Principles and Practice*, Addison-Wesley Publishing Company; Massachusetts, 1990.

Friedhoff, Richard M., and Benzon, William, *Visualization: The Second Computer Revolution*, New York, Harry N. Abrams, Inc., 1989.

Glassner, Andrew, ed., *Graphics Gems*, Academic Press, Inc., Boston, Massachusetts, 1990.

Hill, F.S., Jr., *Computer Graphics*. Macmillan Publishing Company, New York, 1990.

Kirk, David, ed., *Graphics Gems III*, Academic Press, Inc., Boston, Massachusetts, 1992.

Robin, Harry, *The Scientific Image: from cave to computer*, Harry N. Abrams, Inc., New York, 1992.

Rogers, David F., *Procedural Elements for Computer Graphics*, McGraw-Hill Book Company, New York, 1985.

Rogers, David F. and Adams, J.Alan, *Mathematical Elements for Computer Graphics*, 2nd Ed., New York, McGraw-Hill Book Company, 1990.

SIGGRAPH Conference Proceedings, Association for Computing Machinery, Inc.: A Publication of ACM SIGGRAPH, New York, various years.

Tufte, Edward, *The Visual Display of Quantitative Information*, Graphics Press, Cheshire, Connecticut, 1983.

Other sources of information

For additional ideas, consult the "DX Repository," available through anonymous FTP (`ftp.tc.cornell.edu.` in directory `pub/Data.Explorer`), and gopher (`ftp.tc.cornell.edu.` port 70). This public software resource includes information and visual programs contributed by Data Explorer users from around the world. We encourage you to contribute your innovations and ideas to the Repository, in the form of new modules, macros, visual programs, and tips and tricks you discover as you learn and master Data Explorer.

On the Internet, the newsgroup `comp.graphics.apps.data-explorer` is used by customers around the word to share information and ask questions. This newsgroup is also followed by Data Explorer developers.

If you have access to the World Wide Web, you can find the Data Explorer home page at `http://www.almaden.ibm.com/dx/`.

Chapter 1. Overview

The Data Explorer libraries allow you to use Data Explorer functionality in a variety of ways. Depending on the task you want to accomplish, the way you use the libraries will vary. Figure 1 shows the Data Explorer architecture.

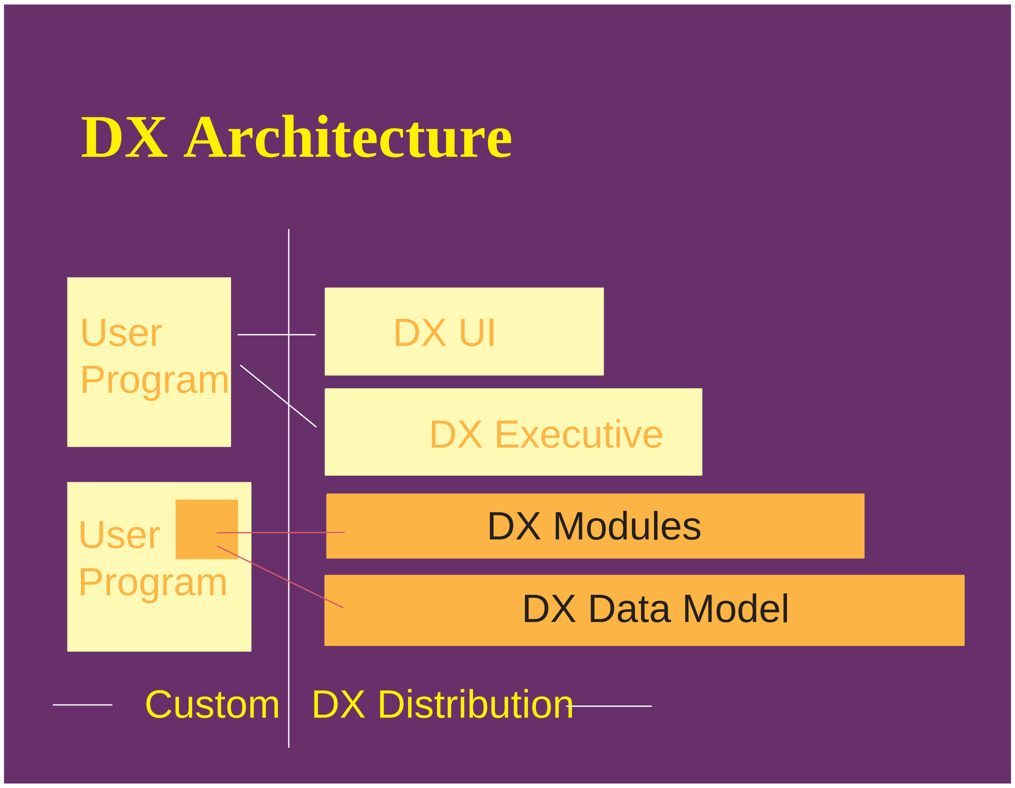


Figure 1. Data Explorer architecture

The Data Explorer User Interface and Data Explorer Executive are the two programs you use when you create and execute visual programs. Underlying the Data Explorer Executive is a collection of modules, which in turn use the Data Explorer data model to manipulate all of the objects in a program (data sets, isosurfaces, images, etc.). The Data Explorer architecture is described in more detail in Chapter 1, “Overview” on page 1 in *IBM Visualization Data Explorer User’s Guide*.

If you want to write a module to use in the Data Explorer Visual Program Editor, or in the scripting language, you will probably use the libDXlite.a library. This library contains all of the Data Explorer data model routines which allow you to query, manipulate, and create data model objects. Much of this reference book concerns this common use of the Data Explorer library routines. You should be familiar with the material in Chapter 11, “Working with Data Model Objects” on page 95, which discusses how to use the Data Explorer data model routines. You may also want to incorporate one or more of the existing Data Explorer modules into your own module, or use some of the high level data processing functions, such as interpolation. In this case you would need to use the libDXcallm.a library. DXCallModule is discussed in 12.10, “Module Access” on page 127.

Chapter 2 through Chapter 10 show you a number of examples of modules, including import filters and various data manipulation modules. These examples are supported by `.c` files and makefiles in `/usr/lpp/dx/samples/program_guide`. If you wish to incorporate routines from the libDXcallm.a library, simply change the makefiles to link to this library instead of to libDXlite.a. All of the routines in libDXcallm.a are described in Appendix C, “Data Explorer Library Routines” on page 183, while the subset of routines available in libDXlite.a is listed in Appendix B, “Data Explorer Data Model Library: DXlite Routines” on page 181. When you write a module for use in Data Explorer, the Data Explorer Executive is still the process which “owns main”. Your module is simply incorporated into Data Explorer. Your module can be directly built in to the Data Explorer Executive (inboard module), run as a separate process (outboard module), or loaded into the Data Explorer executive at runtime (runtime-loadable module). Each of these is discussed in Chapter 10, “Making a Module Work” on page 79. Using runtime-loadable modules is in general the preferred option.

1.1 Writing a Stand-alone Program Using the Data Explorer Data Model

You may also want to write a stand-alone program which uses the Data Explorer data model. For example, you may want to write a data filter which processes data, writing it out to a file in Data Explorer format using DXExportDX. In this case, your stand-alone program “owns main” and simply links in Data Explorer data model routines, which are listed in Appendix B, “Data Explorer Data Model Library: DXlite Routines” on page 181, and discussed in Chapter 11, “Working with Data Model Objects” on page 95. Graphically, this is represented by the lower “User Program” in Figure 1 on page 2, which embeds Data Explorer data model routines into the user's program.

1.2 Writing a Stand-alone Program Using Data Explorer Modules

You may want to write a stand-alone program which directly uses Data Explorer modules. You would link to the libDXcallm.a library, and use DXCallModule to call individual Data Explorer. In this case, as with the previous one, your stand-alone program “owns main”. Note that you can do complete visualization programs in this way, from Import to Isosurface to Display from within your own program. However, you will **not** be getting the functionality of the Data Explorer Executive in this case, including cache management, and control of execution order. You will, in addition, be responsible for deleting objects when you are finished using them. Note that with the SuperviseWindow and SuperviseState modules (see “SuperviseWindow” on page 336 and “SuperviseState” on page 332 in *IBM Visualization Data Explorer User’s Reference*), direct manipulation within the Image window is available without the Image tool, so that a program using DXCallModule can provide direct interaction with objects.

Examples of stand-alone programs including `.c` files and makefiles which use the CallModule library can be found in `/usr/lpp/dx/samples/callmodule`. Graphically, this is represented by the lower “User Program” in Figure 1 on page 2, which embeds Data Explorer module routines into the user's program.

1.3 Controlling the Data Explorer Executive or User Interface from a Separate Program

You may want to write a program which controls the Data Explorer Executive. For example, you could write your own user interface, providing a custom “look and feel”, and send Data Explorer script language commands to the Data Explorer executive. In this case you would get all of the functionality provided by the executive (cache management, control of execution order, and object management). You could also directly control the Data Explorer User Interface from a separate program, loading and executing visual programs. For example, you may wish to fire up Data Explorer with a “canned” visualization program once a simulation is complete, with parameters within the visual program preset to particular values.

Graphically, both of these are represented by the upper “User Program” in Figure 1 on page 2, which controls the Data Explorer Executive or User Interface from the user's program. The libDXL.a library (DXLink) provides this functionality, and is discussed in Chapter 16, “DXLink Developer's Toolkit” on page 157. Examples of DXLink programs can be found in `/usr/lpp/dx/samples/dxlink`.

With the functionality provided by SuperviseWindow and SuperviseState (see “SuperviseWindow” on page 336 and “SuperviseState” on page 332 in *IBM Visualization Data Explorer User’s Reference*), your program does not need the Image tool (which is provided only within the Data Explorer User Interface) in order to provide direct user interaction in the image window. Thus a custom GUI communicating only with the Data Explorer Executive can implement all of the user-interaction provided by the Data Explorer User Interface. Examples of custom direct interactors can be found in `/usr/lpp/dx/samples/supervise`; while these examples are demonstrated using the Data Explorer User Interface, there is no necessity that they do so, as all of the modules used in these examples (SuperviseWindow, SuperviseState, and Display, in particular) are available directly from the Data Explorer Executive.

Chapter 2. Writing a Simple Module

This chapter discusses the basics of writing a simple module for Data Explorer. Subsequent chapters cover some typical types of modules you might want to write. Although Data Explorer modules support a broad range of data and connections types, your module need support only those types it can be expected to encounter. Moreover, it is not necessary to manipulate all the components of a Data Explorer Field. The programming examples in later chapters illustrate modules that manipulate particular components (e.g., “data”).

Before writing a module, you should have a general understanding of the Data Explorer data model and be familiar with the way data is carried in Fields, Groups, and components (see 2.3, “Data Explorer Data Model” on page 12). For a detailed treatment of the data model, see Chapter 3, “Understanding the Data Model” on page 15 in *IBM Visualization Data Explorer User’s Guide*. In this manual, Chapter 11, “Working with Data Model Objects” on page 95, summarizes the routines that implement the model.)

Two other important topics are briefly reviewed in subsequent sections:

1. managing the memory allocated to and used by visual programs and their constituent modules (see 2.4, “Memory Management” on page 13) and
2. the Data Explorer execution model (see 2.5, “Data Explorer Execution Model” on page 14).

2.1 Getting Started Writing a Module

To build a module, you must:

1. Define the module’s function and its interface (i.e., its inputs and outputs).
2. Create a *module description file* containing this information.
3. Write the module.
4. Compile and link the module.

Once you have completed these four steps, you can run a version of Data Explorer that incorporates the module.

The Module Builder is a point-and-click interface that facilitates much of this work by creating the files necessary for a module:

- a module description file
- a C-code framework (or template) file
- a makefile.

All you need do is add your own application code to the framework file. (See Chapter 3, “Module Builder” on page 17.)

A module can be added to Data Explorer in one of three forms: *inboard*, *outboard*, or *runtime-loadable*. An inboard module is linked directly into a new Data Explorer executive. An outboard module is a separate executable linked to the Data Explorer routine library and controlled by the executive. It can later be compiled and linked as an inboard module for greater efficiency. A runtime-loadable module can be loaded when Data Explorer is started or while it is running. It is more portable than the inboard module version of the same function and more efficient than the outboard version. See 10.3, “Inboard, Outboard, and Runtime-loadable Modules” on page 85.

2.2 Adding the Hello Module

The procedure in this example follows the 4-step sequence outlined above.

(1) Define the module’s function, inputs, and outputs

The Hello module appends an input string to "hello." The resulting combination is the module’s output. If the input string is **NULL**, the default output is "`hello world.`"

(2) Create a module description file

Data Explorer’s graphical user interface and executive access the module description file to determine the names of the modules and their inputs and outputs.

Note: This type of file is commonly referred to as an “mdf” file (because of its file extension) and is created automatically from user input to the Module Builder, as described in Chapter 3, “Module Builder” on page 17. However, for very simple modules like the one in this example, it is usually easier and quicker to create the file with a text editor.

Parameter names are a part of the module interface that can be seen by the user. In the graphical user interface, parameter names appear in the configuration dialog box and also serve as default names for interactors. In the scripting language, module parameters can be specified by name.

A new module cannot have the name of an existing Data Explorer module (see *IBM Visualization Data Explorer User’s Reference* for a complete list). You should also be aware of the following requirements:

- A Data Explorer module name must be a single alphanumeric string and its first character must be a letter. (Standard Data Explorer module names capitalize the first character of each “word” in a name, as in Attribute and AutoAxes. You may observe this convention or not, as you wish, for the modules you create.)
- The module must be assigned to a tool category—in a **CATEGORY** statement in the module description file. The category can be any of those listed in the Category palette of the VPE window or a new one that you create by naming it in the statement.

In the following example, the mdf file consists of five statements:

```
MODULE Hello
CATEGORY Greetings
DESCRIPTION  Prefixes “hello” to the input string
INPUT value; string; “world”; input string
OUTPUT greeting; string; prefixed string
```

MODULE	Specifies the module name as Hello.
CATEGORY	Assigns the module to a new, user-specified category (Greetings).
DESCRIPTION	Describes Hello’s purpose: to affix “hello” to the input string.
INPUT	Assigns the name **value** to the input parameter; specifies its parameter type as **string**; specifies its default value as “world”; and describes it as an input string.

OUTPUT Assigns the name **greeting** to the output parameter; specifies its parameter type as **string**; and describes it as a prefixed string.

For details, see 10.1, “Module Description Files” on page 80.

(3) Write the module

Having defined the module in a description file, you can now implement the module with a C-language function like the one shown here.

```
01   #include <dx/dx.h>
02
03
04   Error m_Hello(Object *in, Object *out)
05   {
06     char message[30], *greeting;
07
08     if (!in[0])
09       sprintf(message, "hello world");
10     else {
11       DXExtractString(in[0], &greeting);
12       sprintf(message, "%s %s", "hello", greeting);
13     }
14
15     out[0] = DXNewString(message);
16     return OK;
17   }
```

The **dx.h** file “included” in line 01 contains the definitions of all the Data Explorer library routines. The name of the function that implements a module must consist of the module name (specified in the **MODULE** statement of the description file) prefixed by **m_**. In this case, the function name is **m_Hello**.

When Data Explorer invokes a module, it passes the module two pointers: the first to an array containing the inputs, the second to an array containing the outputs. (See 2.5, “Data Explorer Execution Model” on page 14 for details of parameter passing.)

Because the input parameter of this module is passed to **m_Hello** as an array of pointers, **in[0]** is the **value** parameter. If no argument is specified for **value**, **in[0]** is **NULL**, and the default output (“hello world”) is placed in **message**. If you do specify an argument, a library routine (**DXExtractString**) extracts it from **in[0]**, and **greeting** becomes a pointer to that string. In line 12, **greeting** is appended to “hello,” creating **message**.

Once **message** has been created, the **DXNewString** library routine creates a String Object for the output **out[0]**.

Note: The output of any Data Explorer module must be a Data Explorer Object (such as an Array, Field, or Group). See Table 1 on page 96 for a complete list of Data Explorer Objects.

(4) Compiling and Linking Hello...

...as an inboard module: Copy the following files to the directory you want to work in:

/usr/lpp/dx/samples/program_guide/Makefile_*workstation*
/usr/lpp/dx/samples/program_guide/hello.c
/usr/lpp/dx/samples/program_guide/hello.mdf

Now rename the makefile to **Makefile** (the name of the default makefile) and enter: `make hello`. This command creates an executable that contains the Hello module.

To invoke this executable (from the directory to which the files were copied), enter:

```
dx -mdf ./hello.mdf -exec ./dxexec.
```

This command starts Data Explorer (the **hello.mdf** file tells the graphical user interface about Hello and its inputs and outputs).

You can now run any visual program that uses the Hello module. One such program is **hello.net** in the directory **/usr/lpp/dx/samples/program_guide**.

...as an outboard module: Copy the following files to the directory you want to work in:

/usr/lpp/dx/samples/program_guide/Makefile_*workstation*
/usr/lpp/dx/samples/program_guide/hello.c
/usr/lpp/dx/samples/program_guide/hello_outboard.mdf

Now rename the makefile to **Makefile** (the name of the default makefile) and enter: `make hello_outboard`. This command creates the executable **hello_outboard**.

To invoke the executable (from the directory to which the files were copied), enter:

```
dx -mdf ./hello_outboard.mdf
```

This command starts Data Explorer (the **hello_outboard.mdf** file tells the graphical user interface about Hello and its inputs and outputs).

You can now run any visual program that uses the Hello module. One such program is **hello.net** in the directory **/usr/lpp/dx/samples/program_guide**.

Note: The mdf file of the outboard module contains one additional statement, **OUTBOARD**, which specifies the executable (`hello_outboard`; see 10.1, “Module Description Files” on page 80). This statement may also specify the name of a host on which to run the executable.

```
    MODULE Hello
    CATEGORY Greetings
    DESCRIPTION  Prefixes “hello” to the input string
    OUTBOARD hello_outboard
    INPUT value; string; “world”; input string
    OUTPUT greeting; string; prefixed string
```

...as a runtime-loadable module: Copy the following files to the directory you want to work in:

/usr/lpp/dx/samples/program_guide/Makefile_*workstation*
/usr/lpp/dx/samples/program_guide/hello.c
/usr/lpp/dx/samples/program_guide/hello_loadable.mdf

Now rename the makefile to **Makefile** (the name of the default makefile) and enter: `make hello_loadable`. This command creates the executable **hello_loadable**.

Note: Runtime-loadable modules are not available for SunOS 4.1 or Data General AViiON.

To invoke the executable (from the directory to which the files were copied), enter:

```
dx -mdf ./hello_loadable.mdf
```

This command starts Data Explorer (the **hello_loadable.mdf** file tells the graphical user interface about Hello and its inputs and outputs).

You can now run any visual program that uses the Hello module. One such program is **hello.net** in the directory **/usr/lpp/dx/samples/program_guide**.

Note: The mdf file of the runtime-loadable module contains one additional statement, **LOADABLE**, which specifies the executable (hello_loadable; see 10.1, "Module Description Files" on page 80).

```
MODULE Hello
CATEGORY Greetings
DESCRIPTION  Prefixes "hello" to the input string
LOADABLE hello_loadable
INPUT value; string; "world"; input string
OUTPUT greeting; string; prefixed string
```

Including Hello in a Visual Program

In this example (Figure 2), the Hello module is not given an input value and therefore uses its default string ("hello world") as output. The Echo module sends the string to Data Explorer's Message window.

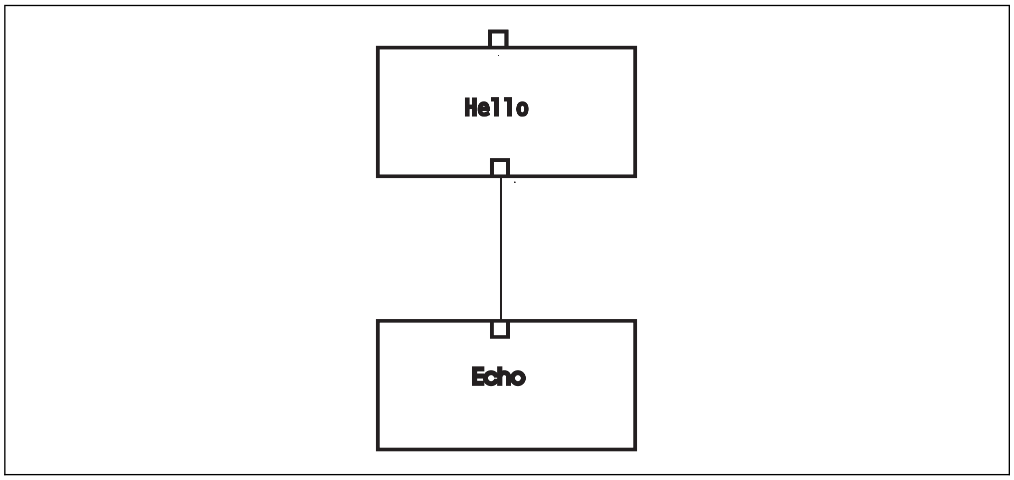


Figure 2. The Hello Module in a Visual Program. The protrusion of the upper tab indicates that the Hello module is using default input. When input is supplied through a connecting "arc," as it is to the Echo module, the input tab folds in.

A visual program that produces the string "hello, how are you?" can be created by:

- Using the visual program in Figure 2 and entering the string ", how are you?" as the argument of the **value** input parameter of the Hello module's configuration dialog box, or
- Creating a visual program (Figure 3 on page 10) in which a string interactor stand-in provides the input (previously entered in the interactor by the user).

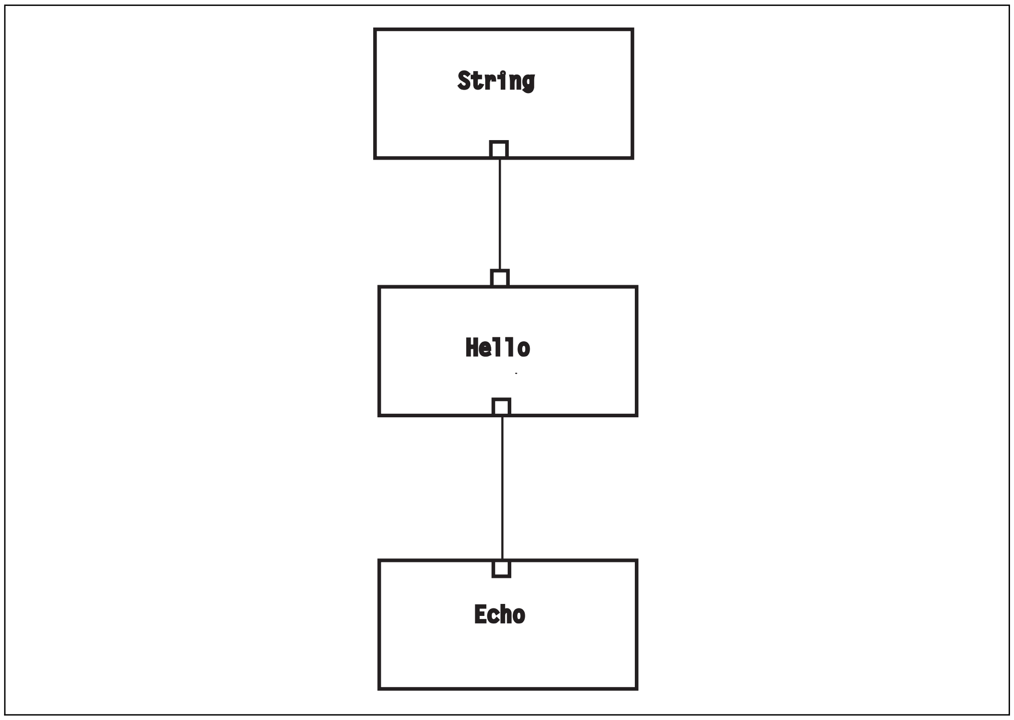


Figure 3. The Hello Module with a String Interactor in a Visual Program. Note that both input tabs are folded in (compare Figure 2).

Using Hello in a Script

In the following example, no input to Hello is provided, so the module produces its default output:

```
a = Hello();

Echo(a);
```

In the next three examples, the user provides input to the Hello module. All three produce the output “hello, how are you?”

Example 1

```
b = Hello(“, how are you?”);
Echo(b);
```

Example 2

```
b = Hello(value = “, how are you?”);
Echo(b);
```

Example 3

```
a = “, how are you?”;
b = Hello(a);
Echo(b);
```

Hello Module with Error Checking

The definition of the Hello module (see 2.2, “Adding the Hello Module” on page 6) contains no error checking code. This omission, of course, is not a recommended practice. In the following version, the Data Explorer routine **DXSetError** reports errors to the user.

```
01   #include <dx/dx.h>
02
03
04   m_HelloErrorChecking(Object *in, Object *out)
05   {
06     char message[30], *greeting, longmessage=NULL;
07
08     if (!in[0])  {
09       sprintf(message, "hello world");
10       out[0] = DXNewString(message);
11     }
12     else {
13       if (!DXExtractString(in[0], &greeting)) {
14          DXSetError(ERROR_BAD_PARAMETER, "value must be a string");
15          goto error;
16       }
17       if (strlen(greeting)<=(28-strlen("hello")) {
18         sprintf(message, "%s %s", "hello", greeting);
19         out[0] = DXNewString(message);
20       }
21       else {
22         longmessage = DXAllocate((strlen("hello")+strlen(greeting)+2)
                                    * sizeof(char));
23         if (!longmessage)
24            goto error;
25         sprintf(longmessage, "%s %s", "hello", greeting);
26         out[0] = DXNewString(longmessage);
27         DXFree((Pointer)longmessage);
28       }
29     }
30     return OK;
31
32   error:
33     DXFree((Pointer)longmessage);
34     return ERROR;
35   }
```

In this example, the return from **DXExtractString** (line 13) is checked. If the routine returns **ERROR**, the error message “value must be a string” is generated and Hello returns **ERROR**.

The combined length of the user-supplied parameter string and “hello” is checked against the length of the buffer. If it exceeds the length, a new buffer is allocated for the output message (and freed before returning). Because **longmessage** is initialized to **NULL**, it can safely be freed on error, even if it has not yet been allocated.

Note: The **m_***module* function should return an error code according to the Data Explorer library standard: **ERROR** for error and **OK** for successful completion. Thus the module entry point would typically be declared by:

```
Error m_module(Object *in, Object *out);
```

To create a version of Data Explorer that includes the HelloErrorChecking module, copy the following files to the directory you want to work in:

/usr/lpp/dx/samples/program_guide/Makefile_*workstation*
/usr/lpp/dx/samples/program_guide/hello_errorchecking.c
/usr/lpp/dx/samples/program_guide/helloerr.mdf

Now rename the makefile to **Makefile** (the name of the default makefile) and enter: `make helloerr`. This command creates an executable that contains the HelloErrorChecking module.

To invoke this executable (from the directory to which the files were copied), enter:

```
dx -mdf ./helloerr.mdf -exec ./dxexec.
```

This command starts Data Explorer (the **helloerr.mdf** file tells the graphical user interface about HelloErrorChecking and its inputs and outputs).

You can now run any visual program that uses the HelloErrorChecking module. One such program is **hello_errorchecking.net** in the **/usr/lpp/dx/samples/program_guide** directory.

2.3 Data Explorer Data Model

Fields are the fundamental informational entities in Data Explorer. A Field contains the components that carry the actual numbers. The “positions” component, for instance, contains the positions of a data set, while the “data” component contains the data values (e.g., temperatures). Groups are higher-level structures and may consist of Fields or other Groups (see Table 1 on page 96).

Generally, a component consists of an Array of data, information describing the data (i.e., its type, dimensionality, or both), and a name associated with the component. Standard components include “positions,” “data,” and “colors.” The name of a component does not usually imply anything about its characteristics (e.g., its data type or dimensionality).

Module operations typically take place at the Field level and involve changing or creating components. For example:

- The Compute module may create an output Field whose data component is the sum of the “data” components of two input Fields.
- The AutoColor module creates an output Object whose “colors” component is based on the “data” component of the input Field.
- The Isosurface module creates an output Object whose “positions” and “connections” components describe a surface of constant value in the input Object’s data Field.

Modules in Data Explorer are generally required to be pure functions and they must not modify their inputs. Instead, to modify the “data” component of an input Object, a module must first make a copy of the Object. Note that the data model allows the module to copy the structure without copying the data. Some modules (e.g., Isosurface) create a completely new Object for the output (as will be illustrated in later chapters).

Because a module typically works by manipulating the components of a Field, and because its input may be a more complicated Object consisting of Groups, it must often operate recursively. In particular, the efficient use of multiple processors requires that a parallel module must be able to traverse Groups, since partitioning creates a special type of Group called a Composite Field.

For example, a module designed to add a number to every data value in each data component of an input Object first makes a copy of the input Object on which it is to operate. This copy duplicates the structure of the input Object (the hierarchy of Groups and Fields) but not the Arrays containing the values in the components of the Fields. In the worker part of the routine, a new “data” component is created to hold the modified data values (the other components can be shared with the input Object, since they will not be modified).

The worker part of the routine first queries the Object to determine whether it is a Field or a Group:

- If it is a Field, the routine extracts the “data” component of the input Object, creates a new “data” component, places it in the output Object, computes the sum, puts the result in the new component, and returns.
- If it is a Group, the routine extracts each of the group’s children and recursively calls itself for each child to determine whether that child is a Field or a Group, and so on. (See Chapter 4, “Working with Data” on page 31 for an illustration of how a module performs this procedure on an input Object.)

2.4 Memory Management

The executive is responsible for managing Objects successfully returned as output by modules and for the memory allocated to those Objects. Any memory allocated or any Objects created that are not returned as output are the responsibility of the module. For instance, in an unsuccessful execution of a module, no Objects are returned. It is important that the programmer remember this difference when writing a module.

Allocating and Freeing Memory

Memory allocation results from either of the following:

- Calls to **`DXAllocate`** and **`DXAllocateLocal`**.

  In general, allocations resulting from these calls must be freed before returning.

- The creation of a new Object (e.g, by **`DXNewField`** or **`DXNewArray`**).

  On successful return, memory allocated for the use of these routines does *not* usually need to be freed: the Objects returned are managed by the executive, and the module is not responsible for their deletion. However, in case of error, no output Objects are returned, and the module is responsible for deleting all Objects created.

  When a Field is placed in a Group, the Field is deleted when the Group is deleted; and, on error, only the Group should be deleted. Similarly, an Array placed in a Field is deleted when the Field is deleted. For that reason it is often helpful to set the pointer of a Field or Array to **`NULL`** after placing it in a higher-level Object. The Field or Array can then be safely deleted on error, regardless of whether it has been placed in a higher-level Object.

Reference Counts

In Data Explorer, reference counts typically require no special action from the user. Thus modules seldom need to call DXReference for any reason, and they usually call DXDelete only to clean up Objects after an error. Note the following:

- All Objects are created with a reference count of zero (0).
- If you call DXDelete on an Object having a reference count of 0 or 1, the Object is invalidated and the space is freed. If the reference count is greater than 1, DXDelete simply decrements the count by 1 and returns.
- If an Object is incorporated in another Object by a call to DXSetComponentValue or DXSetMember, its reference count is incremented to 1.

  This means you can create Array Objects and use DXSetComponentValue to add them to a Field as a component, without having to call DXDelete: the Array will be deleted when the Field itself is deleted. (However, if an error occurs before you add the Array to the Field, you must call DXDelete.)
- Objects returned as output from your module should not be referenced.
- New Objects that are not part of another Object will have a reference count of 0. The executive increments the reference counts for outputs used by other modules and deletes the Objects when memory space is needed.

2.5 Data Explorer Execution Model

The executive executes your module by calling the C function that you provide and passing it two arguments: one pointer to an Array for the input Object(s); another to an Array for the output Object(s) the module creates. In each case, the size of the Array is defined by the corresponding mdf file and the number of input and output statements found there.

If a parameter is not specified by the user when the module is called, the corresponding element of the input Array is set to **NULL**.

As noted earlier, Data Explorer modules are generally required to be pure functions, producing the same results from the same inputs. The reason for this requirement is that the executive caches the results of module execution and does not reexecute the module (given the same input) if the previously computed result is still in the cache.

In particular, modules must not maintain state (e.g., by saving values in global or static variables). Indeed, it may be impossible for a module to maintain state in a multiprocessor environment, since it may execute on a different processor each time. However, a module may use the cache to maintain information that makes succeeding invocations more efficient, provided that it maintains pure-function semantics. (For more information, see 12.5, “Cache” on page 121.)

Outboard modules whose **PERSISTENT** flag is set (see page 80) may maintain state, but they are still required to maintain pure-function semantics: the executive might not execute a module if its inputs have not changed and the results are still in the cache. Modules that have no outputs are executed every time a visual program using them is run, regardless of whether or not their inputs have changed. See also Chapter 4, “Data Explorer Execution Model” on page 37 in *IBM Visualization Data Explorer User’s Guide*.

For Future Reference

Although you must supply a C function for Data Explorer to call, it is still possible to write the bulk of a module in FORTRAN: Write a C “wrapper” that (1) extracts the data from the input Object (using the programming interface described in this manual) and (2) passes the data to the FORTRAN subroutine.

You should be aware of the following when writing a module that interfaces to a FORTRAN routine:

- FORTRAN variables are always called by reference. For example, if you are using the Module Builder you need to modify the worker routine so that it passes the addresses of the parameters.
- Since FORTRAN routines do not return a value, the ERROR (or OK) return value must be a parameter.
- Depending on the compiler, it may be necessary to affix an underscore to the name of the FORTRAN routine and to lowercase the name of the worker routine.
- Passing strings from C to a FORTRAN routine may require passing both the string length and the string pointer. Consult the appropriate compiler manual.

Chapter 3. Module Builder

The Module Builder is a point-and-click interface for creating a module from user-supplied information.

The next several sections describe the general structure of modules built with the Module Builder; its dialog box; and examples of the "worker routine"—the interface between a module and the user-supplied application code.

3.1 Overview

From specifications supplied by the user, the Module Builder generates three files:

1. a module description file (with the extension **`.mdf`**);
2. a C-code framework file that implements the structure of the module; and
3. a makefile.

The C-code framework file itself consists of three major routines:

- The first routine checks input parameters and creates output Objects. It is named by prefixing **`m_`** to the module name (e.g., the entry point for a module named Example is **`m_Example`**).
- The second routine traverses hierarchically defined data Objects. When this routine (**`Traverse`**) encounters a Data Explorer Field Object, it calls the third routine.
- The third routine (**`doLeaf`**) creates a "worker routine" as an interface to the user's application-specific code: it extracts the input arguments, sets up the outputs, and calls the user-supplied code. (See 3.7, "Worker Routine" on page 26 and "Worker Routine Examples" on page 28.). The worker routine appears at the end of the C-code framework file.

To complete a customized module, the user need only:

add the application-specific code to the worker routine (after the line "User's code goes here" at the end of the C-code framework file) by:
using an "include" file, *or*
adding the application code directly to the framework file.

Using an include file is recommended because (1) if you rerun the module builder, it will overwrite the **`.c`** file, and (2) it makes replacing your own code easy.

Notes:

1. The Module Builder is designed to accommodate the most common form of customized module, in which the output Object is a copy of the input, but the data component is modified.
2. Other inputs can be constant parameters or other hierarchically defined data Objects (note that the hierarchy of the data Objects must match exactly).

3.2 Creating a Module with the Module Builder: A Summary

To begin a Module Builder session, enter: `dx -builder`. The dialog box (Figure 4 on page 19) consists of a menu bar and two major sections:

- A **`Overall Module Description`** section for defining the module, and
- An **`Individual Parameter`** section for defining the individual input and output parameters.

When all the necessary information has been entered, save it. For new modules, use the **Save as...** option in the **File** pull-down menu (or **Save**, if the specified module name is already known to the system). The saved file has the name of the module and the extension **.mb**.

You can now use the options in the **Build** pull-down menu to create a module description file (**.mdf**), a C-code module framework (**.c**), a makefile (**.make**), or all three together. You can incorporate your own application code in the C-code framework file and reference additional files in the makefile.

Compile the program as follows:

```
make -f filename.make
```

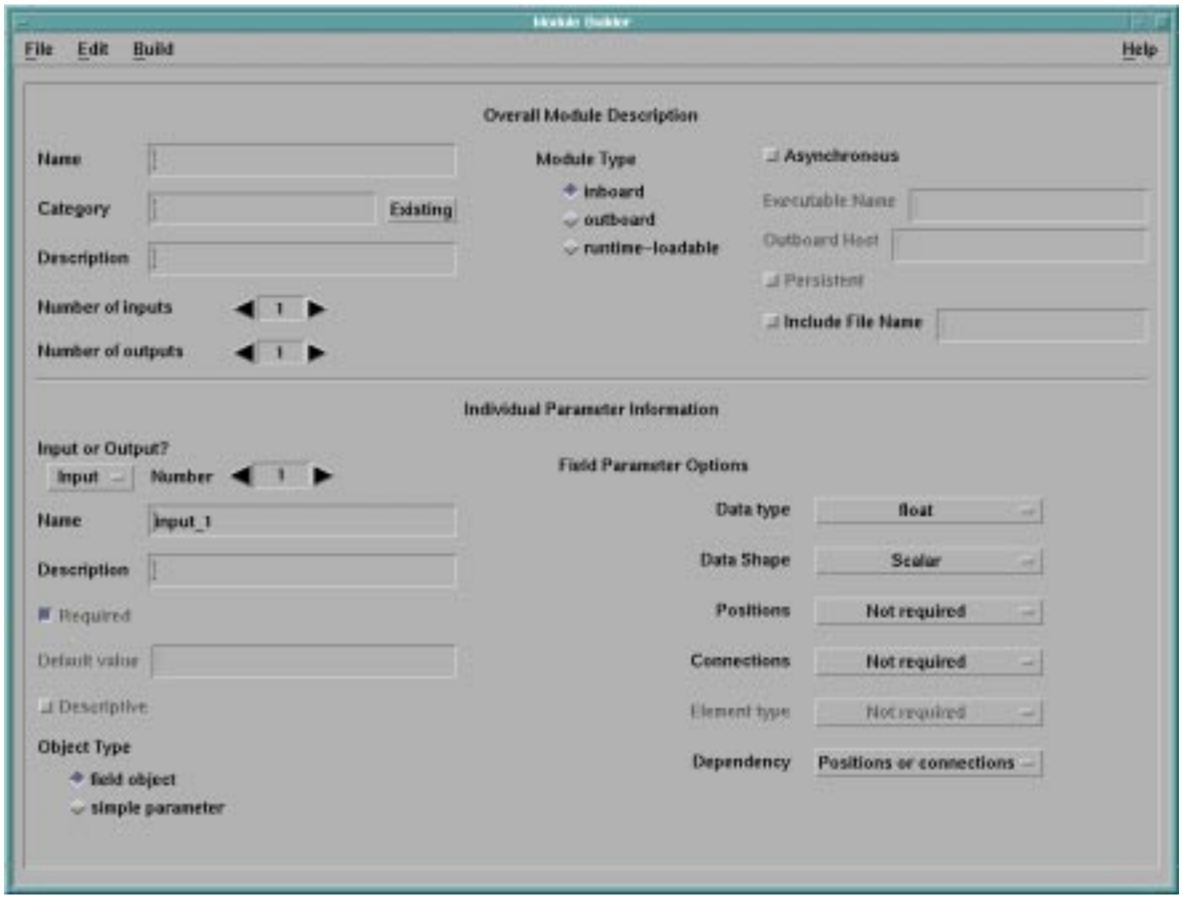
This command creates a customized version of the Data Explorer executive that can use the new module. To run this version in your working directory, enter the following command:

For an *inboard* module:

```
dx -mdf filename.mdf -exec dxexec
```

For an *outboard* or *runtime-loadable* module:

```
dx -mdf filename.mdf
```



Overall Module Description

Name:
Category: [Existing]
Description:
Number of inputs: 1
Number of outputs: 1

Module Type: [✓ inboard] outboard runtime-loadable

Asynchronous
Executable Name
Outboard Host
Persistent
Include File Name

Individual Parameter Information

Input or Output?: Input   Number: 1
Name: input_1
Description:
[✓ Required]
Default value
Descriptive
Object Type: [✓ field object] simple parameter

Field Parameter Options

Data type: float
Data Shape: Scalar
Positions: Not required
Connections: Not required
Element type: Not required
Dependency: Positions or connections

Figure 4. Module Builder Dialog Box. In the **Individual Parameter Information** *section, the* **Input or Output?** *button specifies the kind of parameter being defined, and the associated* **Number** *setting specifies its ordinal ranking (i.e., first, second, etc.). Use the* **Number** *stepper buttons to proceed from one parameter description to the next.*

To create a version of the Data Explorer executive with more than one customized module, you must:

1. concatenate in a single mdf file the module descriptions you want to use;
2. create a makefile that references the combined mdf file as well as all the individual `.c` framework files.
3. compile the program as above.

You can now run the new version with the new mdf file.

3.3 Using the Module Builder: A Quick Walk Through

In this example, you will use the Module Builder to create a simple module that adds a single number to each data value of a set.

Note: This function is supplied with Data Explorer as part of the **Compute** module and is used here only for purposes of illustration.

1. Start the Module Builder with the command `dx -builder`. The Module Builder dialog box appears.
2. Fill in the **Overall Module Description** section as follows:

   Name Type in a name for the module (for this example, use "Add.")

   Category Select a category for the new module, either by clicking on one of the entries in the **Existing** pull-down menu or by typing in a menu name or a name of your choosing.

   Description Type in a short description of the module. This description will appear in the module's configuration dialog box (in the VPE window).

   Number of inputs
   Use the stepper buttons to set the value to "2." The two inputs of the new module will be a data Field and the number to be added to each data value.

   Number of outputs
   The module generates a single output. Use the stepper buttons to set the value to "1" if that is not already the value. The output is the new Field (i.e., the data value plus the added value).

   Outboard Activate the **outboard** toggle button and type in "Add" in the **Executable Name** field. (Leave the other toggle buttons and fields as they are.)

   Note: In this example, the new ("outboard") module is an independent process. In contrast, an "inboard" module is compiled and incorporated as part of the Data Explorer executable; and a runtime-loadable module is compiled independently and loaded into Data Explorer at run time.

3. Fill in the **Individual Parameter Information** section as follows:

   Input Number ◄ 1 ►
   The first input is the data field.

   a. Set the button in the top-left corner of the **Individual Parameter Information** section to **Input** if that is not already the setting.

b. Use the stepper buttons to set **Number** to “1” if that is not already the value.
c. Enter a name for this parameter in the **Name** field (e.g., “data”).
d. Enter a short description in the **Description** field.
e. Confirm that the **Required** toggle button for the first input is activated.
f. Confirm that the **field object** toggle button (below **Object Type**) is activated.
g. On the right-hand side of the **Individual Parameter Information** section, select **float** (“floating point”) and **Scalar** for **Data type** and **Data shape** respectively.
h. Set **Positions** and **Connections** to **Not required** (information about these components is not needed).
i. For this example, since the dependency of the data on positions or connections in the output Object will be the same as it was in the input Object, select **Positions or Connections**.

Input Number ◄ 2 ►

The second input is the value to be added to each data value.

a. Confirm that the button in the top-left corner of the **Individual Parameter Information** section is set to **Input**.
b. Use the stepper buttons to set **Number** to “2”.
c. Enter a name for this parameter in the **Name** field (e.g., “value”).
d. Enter a short description in the **Description** field.
e. Confirm that the **Required** toggle button is deactivated.
f. Specify a default value, such as “0.”
g. If the default value is meant to be descriptive, activate the **Descriptive** toggle button.
h. Under **Object Type**, activate the **simple parameter** toggle button.
i. Now activate the **Scalar** toggle button (below **Type** on the right-hand side).

Output Number ◄ 1 ►

Now describe the output of the module.

a. Change **Input** to **Output** (top-left corner of the **Individual Parameter Information** section). The **Number** setting changes to “1” automatically.
b. Enter a name for this parameter in the **Name** field (e.g., “result”).
c. Enter a short description in the **Description** field.
d. Confirm that the **field object** toggle button (below **Object Type**) is activated.
e. Confirm that **Data type** and **Data Shape** are set to **float** and **Scalar** respectively.
f. Confirm that **Dependency** is set to **Positions or connections**.

You can now create the necessary files:

a. Click on **Build** in the dialog box menu bar and select **Create All** in the pull-down menu.

The Module Builder creates three files for the Add module: `Add.c`, `Add.mdf`, and `Add.make`.

b. To insert the necessary user-specific code in `Add.c`, use an editor. Look for the phrase “User’s code goes here” at the bottom of the file, and type in the following:

```
int i;
float value;

if (value_knt==0)
    value = 0;
else
    value = value_data[0];

for (i=0; i<data_knt; i++) {
    result_data[i] = data_data[i]+value;
}
```

First, variables `i` and `value` are declared. Next, the default value of `0` is enforced by checking whether a number has been passed as the `value` parameter. If `value_knt` is equal to `0`, the user did not pass a number, and `value` is set to `0`. Otherwise, the passed number is used. Finally, the data component of the output is modified by adding `value` to every data item in the input.

Alternatively, you could have put this information in a separate file and used the **Include File Name** option in the **Overall Module Description** section.

To create a version of Data Explorer that uses this module, type

```
make -f Add.make
```

To run Data Explorer using this module, type

```
dx -mdf Add.mdf
```

If you were creating an inboard module instead, the only difference would be in the command to run Data Explorer:

```
dx -mdf Add.mdf -exec ./dxexec
```

3.4 Module Builder: Menu Bar

The pull-down menus of the Module Builder are **File**, **Edit**, **Build**, and **Help**.

File Options

New	Reinitializes the Module Builder.
Open	Opens a specified file (**.mb**) for modification.
Save	Saves the current module specification under the current name (grayed-out unless a name has been provided with Open or Save As...).
Save As...	Saves the current module specifications under a new name.
Quit	Exits the Module Builder.

Edit Options

Comment	Specifies a comment to be associated with the module. This comment will appear in both the resulting C-code and mdf files.
Options...	Specifies one of the following options for a module: • Pin—Run the module on the same processor each time the visual program is executed (in a multiprocessor environment). • Side_Effect—Run the module every time the visual program is executed—because the module produces additional effects (other than its output), such as writing out a log file.

Build Options

Create MDF	Create an mdf file from the current module specifications.
Create C	Create a C-code framework file from the current module specifications.
Create Makefile	Create a makefile from the current module specifications.
Create All	Create all three files from the current module specifications.
Build Executable	Compile and link a version of Data Explorer using this module.
Build and Run DX	Compile, link, and run a version of Data Explorer using this module.

Help Options

The Help options for the Module Builder are the same as those for the Image window and the VPE (as detailed in the User's Guide). You can access this information through the **Using Help...** option of the **Help** pull-down menu.

3.5 Module Builder: Overall Module Description

Use this section to specify the following information.

Name	The name of the module.
Category	The name of the tool category to which the module is to be assigned. This can be the name of an existing category or a name of your choosing. Click on **Existing** (at the right of the text field) for a list of existing categories. If you then click on one of the categories in the list, the category name is automatically inserted in the text field.
Description	A brief description of the module being defined. It appears in a **Description of...** box when the **Description...** button of the module's configuration dialog box is selected.
Number of inputs	The number of inputs to the module.
Number of outputs	The number of outputs from the module.
Module Type	The type of module: inboard, outboard, or runtime-loadable. Specify the type by activating the appropriate toggle button. The default is "inboard."

Notes:

1. Activating the **outboard** toggle button makes available two associated text fields (**Executable Name** and **Outboard Host**) and one toggle button (**Persistent**):

Executable Name	Specifies the name of the module executable.
Outboard Host	Specifies the name of the host machine on which the outboard module is to be run. This parameter defaults to “local host.”
Persistent	Specifies that the outboard module is not be terminated after it produces output.

   For more information on outboard modules, see 10.5, “Compiling, Linking, and Debugging an Outboard Module” on page 86.

2. Activating the **runtime-loadable** toggle button makes available the **Executable Name** text field. For more information, see 10.6, “Compiling, Linking, and Debugging a Runtime-loadable Module” on page 91.

Executable name	See **Module Type** above.
Outboard Host	See **Module Type** above.
Persistent	See **Module Type** above.
Asynchronous	The module may create data asynchronously, in response to events outside of Data Explorer. See 10.2, “Asynchronous Modules” on page 84.
Include File Name	The name of an include file to be inserted in the worker routine.

3.6 Module Builder: Individual Parameter Information Section

Use this section to specify the following information about each input and output parameter.

Input or Output?	The kind of parameter: input or output.
Number	The ordinal number of the input or output being defined: “1” (first), “2” (second), and so on.
Name	The parameter name, which appears in the module’s configuration dialog box and is displayed when the tab of the module stand-in is selected. It may also be used in script mode and appear in the C-code framework file.
Description	A brief description of the parameter being defined. It appears in a **Description of...** box when the **Description...** button of the module’s configuration dialog box is selected.
Required	Specification that the parameter must be set, because no default value is possible. This option is grayed-out for output parameters.
Default value	The default value of the parameter. It appears in the module configuration dialog box and is included in the C-code framework file as a comment.

Note: Implementing the default value is the module writer's responsibility.

Descriptive	Specifies whether the default value is an actual value or a descriptive phrase (e.g., "center of object").
Object Type	This parameter allows the user to specify a Field Object or a simple parameter. Each specification has an associated set of options that are enabled when the appropriate toggle button is activated.

Field Parameter Options

Data type The type of the data associated with the parameter (click on the associated button to display a list of valid types). This information is used in the C-code framework file to check for errors before the user's routine is called.

Data Shape The "shape" of the data associated with the parameter (e.g., scalar). This information is used in the C-code framework file to check for errors before the user's routine is called.

Positions One of the following options controlling information about the positions component of the first input parameter:

- **Not required**—No positions information is passed.
- **Regular**—a compact representation of the positions is passed.
- **Irregular**—An explicit Array of positions is passed.

This option is grayed-out for all but the first input argument.

Connections One of the following options controlling information about the connections component of the first input parameter:

- **Not required**—No connections information is passed.
- **Regular**—a compact representation of the connections is passed.
- **Irregular**—An explicit Array of connections is passed.

This option is grayed-out for all but the first input argument.

Element type The type of connection element expected by the module: lines, quads, cubes, triangles, or tetrahedra. The specification is checked against the connection type of the first input argument before the user's routine is called.

This option is grayed-out unless the connections component is passed in by the Connections option.

Dependency	The dependency of the data component of the input. The specification is checked against the dependency of the input Fields.

Simple Parameter Options

This set of options is enabled and displayed only when the **simple parameter** toggle button is activated (see above).

Type	The type of the parameter, as specified by activating one of 10 toggle buttons.
Vector Length	The length of the vector (given that the parameter type is a vector).

3.7 Worker Routine

At the end of the C-code framework file created by the Data Explorer Module Builder is a “worker routine” that serves as an interface to the user’s application code. The entry point to this routine (i.e., its name) consists of the module name affixed to “_worker” (e.g., the name of the worker routine for the **X** module is **X_worker**).

The Module Builder prepares a parameter list for the worker routine that contains information from the module input parameters, along with pointers to memory for module results. Three examples appear in “Worker Routine Examples” on page 28. In each example, the Module Builder creates a module with two inputs (the first a Field/Group, and the second a Value) and one output (a Field/Group Object).

Positions Specification

If the worker routine needs information about the positions component of the first input parameter, it will use one of two sets of arguments to define that information, depending on whether the request is for regular or irregular positions:

```
Regular positions arguments:

int      p_knt          Total number of positions
int      p_dim          Dimensionality of positions
int     *p_counts       Count of positions along each dimension
float   *p_origin       Origin of regular grid
float   *p_deltas       Delta vectors, p_dim x p_dim

Irregular positions arguments:

int      p_knt          Total number of positions
int      p_dim          Dimensionality of positions
float   *p_positions    Explicit list of positions
```

Connections Specification

If the worker routine needs information about the connections component of the first input parameter, it will use one of two sets of arguments for defining that information, depending on whether the request is for regular or irregular positions:

```
Regular connections arguments:

int      c_knt          Total number of connections elements
int      c_nv           Number of vertices per element
int     *c_counts       Count of vertices along each dimension
```

```
Irregular connections arguments:

int      c_knt          Total number of connections elements
int      c_nv           Number of vertices per element
int     *c_connections  Explicit list of connections elements
```

Input Data

For each input to the module, a count value and a pointer are sent to the worker routine. These arguments are named by appending **_knt** and **_data** respectively to the parameter name given in the **Individual Parameter Information** section of the Module Builder interface. Thus, for an input parameter named **param1**, the worker routine would add the following to its argument list:

```
int param1_knt     The number of elements in the parameter.
type *param1_data  A pointer to the data associated with the parameter.
                   The pointer type is that specified in the Data type
                   field of the Individual Parameter
                   Information section of the
                   Module Builder.
```

For a Field/Group input, these arguments reflect the contents of the “data” component of the leaf. For a Value input (which must be a Data Explorer Array), they reflect the contents of the parameter. Because the parameters are inputs to the module, the arguments are read-only.

Output Data

For each output of the module, a counts value and a pointer are sent to the worker routine. These arguments are named by appending **_knt** and **_data** respectively to the parameter name given in the **Individual Parameter Information** section of the Module Builder dialog box. Thus, for an input parameter named **param2**, the worker routine would add the following to its argument list:

```
int param2_knt     The number of elements in the parameter.
type *param2_data  A pointer to the data associated with the parameter.
                   The pointer type is that specified in the Data type
                   field of the Individual Parameter
                   Information section of the
                   Module Builder.
```

For a Field/Group output, these arguments reflect the contents of the “data” component of the leaf (if the leaf is a Data Explorer Field) or of the array itself (if the leaf is a Data Explorer Array). For a Value input (which must be a Data Explorer Array), they reflect the contents of the parameter. The memory associated with these parameters is not initialized.

Implementing Default Input Parameters

If an input parameter is not provided, the corresponding counts argument of the worker routine for that parameter is zero (0). Implementing the default for the input parameter is the function of the worker routine.

Worker Routine Examples

Figure 5. Worker Routine: Example1_worker. This routine requests no positions or connections.

```
int
Example1_worker(
     int inputObject_knt, float *inputObject_data,
     int inputArgument_knt, float *inputArgument_data,
     int outputObject_knt, float *outputObject_data)

{
 /*
  * The arguments to this routine are:
  *

  * The following are inputs and therefore read-only.  The default
  * values are given and should be used if knt is 0.
  *
  * inputObject_knt, inputObject_data: count and pointer for input "inputObject"
  *                  no default value given.
  * inputArgument_knt, inputArgument_data: count and pointer for input "inputArgument"
  *                  nondescriptive default value is "1.0"
  *
  *  The following are outputs and therefore writable.
  *
  * outputObject_knt, outputObject_data: count and pointer for output "outputObject"
  */

 /*
  * User's code goes here.
  */

}
```

Figure 6. Worker Routine. Example2_worker. This routine requests regular positions and connections.

```
int
Example2_worker(
 int p_knt, int p_dim, int *p_counts, float *p_origin, float *p_deltas,
 int c_knt, int c_nv, int *c_counts,
 int inputObject_knt, float *inputObject_data,
 int inputArgument_knt, float *inputArgument_data,
 int outputObject_knt, float *outputObject_data)
```

```
{
 /*
  * The arguments to this routine are:
  *
  *  p_knt:           total count of input positions
  *  p_dim:           dimensionality of input positions
  *  p_counts:        count along each axis of regular positions grid
  *  p_origin:        origin of regular positions grid
  *  p_deltas:        regular positions delta vectors
  *  c_knt:           total count of input connections elements
  *  c_nv:            number of vertices per element
  *  c_counts:        vertex count along each axis of regular positions grid
  *
  * The following are inputs and therefore read-only.  The default
  * values are given and should be used if knt is 0.
  *
  * inputObject_knt, inputObject_data: count and pointer for input "inputObject"
  *                  no default value given.
  * inputArgument_knt, inputArgument_data: count and pointer for input "inputArgument"
  *                  nondescriptive default value is "1.0"
  *
  *  The following are outputs and therefore writable.
  *
  * outputObject_knt, outputObject_data: count and pointer for output "outputObject"
  */

 /*
  * User's code goes here.
  */

}
```

Figure 7. Worker Routine. Example3_worker. This routine requests irregular positions and connections.

```
int
Example3_worker(
 int p_knt, int p_dim, float *p_positions,
 int c_knt, int c_nv, int *c_connections,
 int inputObject_knt, float *inputObject_data,
 int inputArgument_knt, float *inputArgument_data,
 int outputObject_knt, float *outputObject_data)
```

```
{
 /*
  * The arguments to this routine are:
  *
  *  p_knt:           total count of input positions
  *  p_dim:           dimensionality of input positions
  *  p_positions:     pointer to positions list
  *  c_knt:           total count of input connections elements
  *  c_nv:            number of vertices per element
  *  c_connections:   pointer to connections list
  *
  * The following are inputs and therefore read-only. The default
  * values are given and should be used if knt is 0.
  *
  * inputObject_knt, inputObject_data: count and pointer for input "inputObject"
  *                  no default value given
  * inputArgument_knt, inputArgument_data: count and pointer for input "inputArgument"
  *                  nondescriptive default value is "1.0"
  *

  *  The following are outputs and therefore writable.
  *
  * outputObject_knt, outputObject_data: count and pointer for output "outputObject"
  */

 /*
  * User's code goes here.
  */

}
```

Chapter 4. Working with Data

For modules that manipulate the data component of an Object, positions and connections are often irrelevant. The Statistics module, for example, computes the mean of a data Field regardless of whether the connections are quads or cubes. In fact, it is unnecessary for the Statistics module to examine or access the connections component at all.

The Module Builder is well suited to creating such modules.

4.1 Add Module Example—Add a Number to Every Data Value

The Add module adds a number to every data value in a Field.

Note: This example is for illustration rather than "practice," since its function is already provided by Compute (see "Compute" on page 86 in *IBM Visualization Data Explorer User's Reference*).

The Add module takes two inputs: the first, **data**, is of type **field** and has no default value; the second, **addend**, is of type **scalar**, and has a default value of zero (0).

The Add module has one output: **result**, of type **field**.

(1) Start the Module Builder with the command:

```
dx -builder
```

The Module Builder dialog box appears. Note that the dialog box carries no information, since no module has been specified. (For a simple example of creating a module with the Module Builder, see 3.3, "Using the Module Builder: A Quick Walk Through" on page 20)

(2) Select **Open** from the **File** pull-down menu. An **Open a Module Builder file...** dialog box appears.

(3) Read in `/usr/lpp/dx/samples/program_guide/add.mb` as follows:

- Type the full path name in the **Filter** field of the dialog box.
- Click on (in sequence):
  - the **Filter** button
  - the name of the file in the **Files** field
  - the **OK** button.

  Information describing the inputs and output of the module (extracted from the `add.mb` file) appears in the Module Builder dialog box. (Of course, if you were creating this module from scratch, you would fill in the information yourself.)

(4) Save the .mb file to a writable directory (use **Save As...** in the **File** pull-down menu).

(5) Select **Create All** from the **Build** pull-down menu of the dialog box. This option creates three files for the module: `add.c`, `add.mdf`, and `add.make`.

(6) Implement the Add function.

Use an editor to add the following lines after "User's code goes here," near the end of the `add.c` file:

```
   int i;
   float value;

   /* first implement the default for addend of 0 */
   if (addend_knt == 0)
      value = 0;
   else
      value = *addend_data;

   /* add addend to each value in the data field */
   for (i=0; i < result_knt; i++) {
      result_data[i] = data_data[i] + value;
   }
   return 1;
```

The file `/usr/lpp/dx/samples/program_guide/add.c` contains a completed version of this program.

(7) To create a version of Data Explorer that includes the Add module, enter the command:

```
make -f add.make dxexec
```

(You have now created an executable that contains the Add module.)

(8) To invoke this version, enter:

```
dx -mdf ./add.mdf -exec ./dxexec
```

This command starts Data Explorer (the **add.mdf** file tells the graphical user interface about Add and its inputs and outputs). The executable dxexec invoked here is the one created in Step 6.

(9) With this version of Data Explorer you can now run any visual program that uses the Add module. One such program is `/usr/lpp/dx/samples/program_guide/add.net`

4.2 Add2 Module Example—Add Two Data Fields

This module adds together the data components of two input Fields. Thus, one of its functions is to ensure that the hierarchies of the two input Objects match one-to-one.

The Add2 module takes two inputs: **field1** and **field2**. Each is of type **field** and has no default value.

The Add2 module has one output, **result**, of type **field**.

Repeat Steps (1) through (5) of the first example (see 4.1, “Add Module Example—Add a Number to Every Data Value” on page 32), using the file name “add2” instead of “add.” Step (5) will produce files `add2.c`, `add2.mdf`, and `add2.make`.

(6) Implement the Add2 function. Use an editor to add the following lines after “User’s code goes here,” near the end of the `add2.c` file:

```
   int i;

   /* first check that the lengths of the data buffers match */
   if (field1_knt != field2_knt) {
      DXSetError(ERROR_INVALID_DATA,"data components do not match");
      return 0;
   }

   for (i=0; i < field1_knt; i++)
      result_data[i] = field1_data[i] + field2_data[i];

   return 1;
}
```

The file `/usr/lpp/dx/samples/program_guide/add2.c` contains a completed version of this program.

(7) To create a version of Data Explorer that includes the Add2 module, enter the command:

```
make -f add2.make dxexec
```

(You have now created an executable that contains the Add2 module.)

(8) To invoke this version, enter:

```
dx -mdf ./add2.mdf -exec ./dxexec
```

This command starts Data Explorer (the **`add2.mdf`** file tells the graphical user interface about Add2 and its inputs and outputs). The executable dxexec invoked here is the one created in Step 6.

(9) With this version of Data Explorer you can now run any visual program that uses the Add2 module. One such program is `/usr/lpp/dx/samples/program_guide/add2.net`

4.3 Add2Invalid Module Example—Manipulate Invalid Data

The Data Explorer data model makes it possible to identify invalid input (position and connections elements) and mark the resulting output as “invalid.” (see 13.3, “Invalid Data” on page 133). Invalid elements (and the data associated with them) are ignored by Data Explorer modules.

In the example given here, the Add2Invalid module processes two input data components. If either of the two data values is invalid, the resulting sum is treated as invalid. The routines that support this function check for matching data types, matching dependencies, missing Fields, and so on.

Repeat Steps (1) through (5) of the first example (see 4.1, “Add Module Example—Add a Number to Every Data Value” on page 32), using the file name “add2invalid” instead of “add.” Step (5) will produce files `add2invalid.c`, `add2invalid.mdf`, and `add2invalid.make`.

(6) Implement the Add2Invalid function. Because this module uses routines for handling invalid data, the necessary modifications of the **`.c`** file are more extensive than those required for the preceding examples.

As written, the `add2_invalid.c` file passes only the data component to the lowest-level routine (Add2Invalid_worker); it does not pass information about the data's validity. The solution is to modify the doLeaf routine, rather than the worker routine. The doLeaf routine has access to all the components of an input or output Field and not to just the data component.

In the routine doLeaf, starting at the comment "Call the user's routine. Check the return code." insert the following:

```
  /* create invalid component handles for each input field */
  inv_handle1 = DXCreateInvalidComponentHandle(in[0], NULL,
                                              src_dependency);
  inv_handle2 = DXCreateInvalidComponentHandle(in[1], NULL,
                                              src_dependency);

  /* the loop that actually adds the data components.
   * if either of the two input data values is invalid, then the
   * output is marked invalid, and set to the value 0
   */
  out_ptr = (float *)out_data[0];
  in1_ptr = (float *)in_data[0];
  in2_ptr = (float *)in_data[1];
  for (i=0; i<out_knt[0]; i++) {
    if (DXIsElementValid(inv_handle1, i) &&
        DXIsElementValid(inv_handle2, i)) {
      *out_ptr =  *in1_ptr + *in2_ptr;
    }
    else {
      *out_ptr = 0.0;
      DXSetElementInvalid(inv_handle1, i);
    }
    out_ptr++;
    in1_ptr++;
    in2_ptr++;
  }

  /* the invalid-component-handle information is added to the output field */
  if (!DXSaveInvalidComponent((Field)out[0], inv_handle1))
     goto error;
  DXFreeInvalidComponentHandle(inv_handle1);
  DXFreeInvalidComponentHandle(inv_handle2);

  return OK;

error:
  return ERROR;
```

(7) Remove the call to Add2Invalid_worker: it is not needed. All of the data processing code has been added to doLeaf.

(8) Insert the following declarations at the top of the routine doLeaf:

```
  InvalidComponentHandle inv_handle1, inv_handle2;
  float *out_ptr, *in1_ptr, *in2_ptr;
```

The file `/usr/lpp/dx/samples/program_guide/add2invalid.c` contains a completed version of this program.

(9) To create a version of Data Explorer that includes the Add2Invalid module, enter the command:

```
make -f add2invalid.make dxexec
```

(You have now created an executable that contains the Add2Invalid module.)

(10) To invoke this version, enter:

```
dx -mdf ./add2invalid.mdf -exec ./dxexec
```

This command starts Data Explorer (the **add2invalid.mdf** file tells the graphical user interface about Add2Invalid and its inputs and outputs). The executable dxexec invoked here is the one created in Step 8.

(11) With this version of Data Explorer you can now run any visual program that uses the Add2Invalid module. One such program is `/usr/lpp/dx/samples/program_guide/add2_invalid.net`

Chapter 5. Working with Positions

The following examples illustrate the manipulation of the “positions” component of a data Field. In these examples it is not necessary to access the “data” component: the data value at a particular position has no effect on the output of the module. This example conforms to the general principle, that the only components of a Field that need to be accessed are those required for the module’s function.

5.1 MakeX Module Example—Create New Positions

The MakeX module places an “x” at every position in an input Field. MakeX differs from the Add module (see 4.1, “Add Module Example—Add a Number to Every Data Value” on page 32) in that, instead of simply modifying a component of the input Field, it creates new positions and connections components.

The MakeX module takes two inputs: the first, **`data`**, is of type **`field`** and has no default value; the second, **`size`**, is of type **`float`**, and has a default value of 1.

The MakeX module has one output: **`result`**, of type **`field`**.

(1) Start the Module Builder with the command:

```
dx -builder
```

The Module Builder dialog box appears. Note that the dialog box carries no information, since no module has been specified. (For a simple example of creating a module with the Module Builder, see 3.3, “Using the Module Builder: A Quick Walk Through” on page 20)

(2) Select `Open` from the **`File`** pull-down menu. An **`Open a Module Builder file...`** dialog box appears.

(3) Read in `/usr/lpp/dx/samples/program_guide/makex.mb` as follows:

- Type the full path name in the **`Filter`** field of the dialog box.
- Click on (in sequence):
  - the **`Filter`** button
  - the name of the file in the **`Files`** field
  - the **`OK`** button.

  Information describing the inputs and output of the module (extracted from the `makex.mb` file) appears in the Module Builder dialog box.

(4) Save the .mb file to a writable directory (use **`Save As...`** in the **`File`** pull-down menu).

(5) Select Create All from the **`Build`** pull-down menu of the dialog box. This option creates three files for the module: `makex.make`, `makex.c`, and `makex.mdf`.

(6) Implement the MakeX function. As in the example of Add2Invalid (see 4.3, “Add2Invalid Module Example—Manipulate Invalid Data” on page 34), the MakeX module needs access to the input Object at a higher level than that provided by the MakeX_worker routine. Consequently, the addition of new user code includes a modification of the routine doLeaf as well.

Use an editor to add the following lines (after extracting the positions Array with **`DXGetArrayData`**):

```
     . . .

     p_position = (float *)DXGetArrayData(array);
     if (! positions)
        goto error;
   }

  /* New User code starts here */

  /*
   * Make the new positions array for the output. The positions are
   * 3-dimensional.
   */
     positions = DXNewArray(TYPE_FLOAT, CATEGORY_REAL, 1, 3);
     if (! positions)
        goto error;

  /*
   * Check that the input positions are 3-dimensional:
   */
     if (p_dim != 3) {
        DXSetError(ERROR_INVALID_DATA,"input positions must be 3-dimensional");
        goto error;
     }

  /*
   * Allocate space to the new positions array. Four positions are needed
   * for every input position (the four points making up the "x").
   */
     if (! DXAddArrayData(positions, 0, 4*p_knt, NULL))
        goto error;

  /* Get a pointer to the output positions. */
     out_pos_ptr  = (Point *)DXGetArrayData(positions);

  /* Make a connections component for the output. The connections are
   * 2-dimensional (lines).
   */
     connections = DXNewArray(TYPE_INT, CATEGORY_REAL, 1, 2);

  /* Allocate space to the new connections array. There are two lines for
   * each input position.
   */
     if (! DXAddArrayData(connections, 0, 2*p_knt, NULL))
        goto error;
     DXSetAttribute((Object)connections, "element type",
                    (Object)DXNewString("lines"));

  /* Get a pointer to the new connections. */
     conn_ptr = (Line *)DXGetArrayData(connections);

  /* Get the size of the "x" */
     DXExtractFloat(in[1], &size);
```

```
/* Now "draw" the x's */
   for (i=0; i< p_knt; i++) {
       inpoint = DXPt(p_positions[3*i], p_positions[3*i+1], p_positions[3*i+2]);
       out_pos_ptr[4*i]   = DXPt(inpoint.x - size, inpoint.y, inpoint.z);
       out_pos_ptr[4*i+1] = DXPt(inpoint.x + size, inpoint.y, inpoint.z);
       out_pos_ptr[4*i+2] = DXPt(inpoint.x, inpoint.y - size, inpoint.z);
       out_pos_ptr[4*i+3] = DXPt(inpoint.x, inpoint.y + size, inpoint.z);

       conn_ptr[2*i] = DXLn(4*i, 4*i+1);
       conn_ptr[2*i+1] = DXLn(4*i+2, 4*i+3);
   }

/* Clean up; we're about to significantly modify the positions and connections
 */
   DXChangedComponentStructure((Field)out[0],"positions");
   DXChangedComponentStructure((Field)out[0],"connections");

/* Now place the new positions and connections in the output field */
   DXSetComponentValue((Field)out[0], "positions", (Object)positions);
   positions = NULL;
   DXSetComponentValue((Field)out[0], "connections", (Object)connections);
   connections = NULL;

/* Finalize the field */
   DXEndField((Field)out[0]);

/* return */
   return OK;
error:
   DXDelete((Object)positions);
   DXDelete((Object)connections);
   return ERROR;
```

(7) Remove the call to MakeX_worker: it is not needed. All of the data processing code has been added to doLeaf (Step 5).

(8) Insert the following declarations after the first block of code in the doLeaf routine:

```
  /* User added declarations */
  Point *out_pos_ptr, inpoint;
  Array connections=NULL, positions=NULL;
  Line *conn_ptr;
  float size;
```

The file `/usr/lpp/dx/samples/program_guide/makex.c` contains a completed version of this program.

(9) To create a version of Data Explorer that includes the MakeX module, enter the command:

```
make -f makex.make dxexec
```

(You have now created an executable that contains the MakeX module.)

(10) To invoke this version, enter:

```
dx -mdf ./makex.mdf -exec ./dxexec
```

This command starts Data Explorer (the **makex.mdf** file tells the graphical user interface about MakeX and its inputs and outputs). The executable dxexec invoked here is the one created in Step 8.

(11) With this version of Data Explorer you can now run any visual program that uses the MakeX module. One such program is `/usr/lpp/dx/samples/program_guide/makex.net`

5.2 MakeXEfficient Module Example—Create New Positions

The preceding example module, MakeX, manipulates regular (compactly encoded) positions less efficiently than it might. Note that the first call to **DXGetArrayData** in the file `makex.c` expands the compact coding of regular positions. MakeXEfficient eliminates this expansion.

MakeXEfficient has the same two inputs as MakeX: **data**, is of type **field** and has no default value; **addend**, is of type **size**, and has a default value of 1.

MakeXEfficient has the same output as MakeX: **result** is of type **field**.

Repeat Steps (1) through (5) of the preceding example (see 5.1, "MakeX Module Example—Create New Positions" on page 38), using the file name "makexeff" in place of "makex." Step (5) will produce files `makexeff.c`, `makexeff.mdf`, and `makexeff.make`.

(6) Implement the MakeXEfficient function. MakeXEfficient uses Array-handling routines like those described in "Array Handling" on page 102. But the main differences from MakeX are a handle for manipulating the input-positions Array, and a scratch buffer to hold the coordinates of a single position (three floating-point numbers in this example). Note that there is no call to DXGetArrayData for a pointer to the input-positions Array, thereby avoiding inefficiencies by not expanding regular positions.

Edit the `makexeff.c` file and replace the line

```
p_positions = (float*) DXGetArrayData(array)
```

with the following:

```
if (!(handle = DXCreateArrayHandle(array)))
   goto error;

scratch = DXAllocate(3*sizeof(float));
if (!scratch)
  goto error;
```

Another necessary code change is one inside the loop labeled "Now "draw" the x's" in the **.c** file—a call to the DXIterateArray routine to access the current position. Add the following pair of lines after the comment in the loop:

```
     in_ptr = (float *)DXIterateArray(handle, i, in_ptr, scratch);
     inpoint = DXPt(in_ptr[0], in_ptr[1], in_ptr[2]);
```

Of course, the handle and the scratch buffer have to be freed at some point. Add the following lines before the MakeXEfficient_worker code near the end of the file:

```
/* Delete scratch and handle */
   DXFree((Pointer)scratch);
   DXFreeArrayHandle(handle);

/* return */
   return OK;
error:

/* Delete scratch and handle */
   DXFree((Pointer)scratch);
   DXFreeArrayHandle(handle);
   return ERROR;
```

Add the following lines after the first block of code in the doLeaf routine:

```
  /* User-added declarations */
  float *scratch, *in_ptr, size;
  Point inpoint, *out_pos_ptr;
  ArrayHandle handle;
  Array connections;
  Line *conn_ptr;
```

The file `/usr/lpp/dx/samples/program_guide/makexeff.c` contains a completed version of this program.

(7) To create a version of Data Explorer that includes the MakeXEfficient module, enter the command:

```
make -f makexeff.make dxexec
```

(You have now created an executable that contains the MakeX module.)

(8) To invoke this version, enter:

```
dx -mdf ./makexeff.mdf -exec ./dxexec
```

This command starts Data Explorer (the **`makexeff.mdf`** file tells the graphical user interface about MakeX and its inputs and outputs). The executable dxexec invoked here is the one created in Step 6.

(9) With this version of Data Explorer you can now run any visual program that uses the MakeXEfficient module. One such program is `/usr/lpp/dx/samples/program_guide/makex_efficient.net`

Chapter 6. Working with Connections

The module in this example manipulates both the data and the connections components of an input Field (the modules in Chapter 4, “Working with Data” on page 31 and Chapter 5, “Working with Positions” on page 37, did not require a connections component). Modules that perform interpolation need information about the interpolation elements (connections). For example, the Isosurface module uses different interpolation methods according to the type of connection.

6.1 AverageCell Module Example—Average the Data Values of All Neighbors

The AverageCell module computes, for each cell, the average of the data values of that cell and all its neighbors.

Note: This module works only on data that is cell-centered (i.e., connection-dependent) and has connections of type “quads.”

The AverageCell module takes one input: **`input`**, of type **`field`**, which has no default value. The module has one output: **`output`**, of type **`field`**.

(1) Start the Module Builder with the command:

```
dx -builder
```

The Module Builder dialog box appears. Note that the dialog box carries no information, since no module has been specified. (For a simple example of creating a module with the Module Builder, see 3.3, “Using the Module Builder: A Quick Walk Through” on page 20).

(2) Select **`Open`** from the **`File`** pull-down menu. An **`Open a Module Builder file...`** dialog box appears.

(3) Read in `/usr/lpp/dx/samples/program_guide/averagecell.mb` as follows:

- Type the full path name in the **`Filter`** field of the dialog box.
- Click on (in sequence):
  - the **`Filter`** button
  - the name of the file in the **`Files`** field
  - the **`OK`** button.

  Information describing the inputs and output of the module (extracted from the `averagecell.mb` file) appears in the Module Builder dialog box.

(4) Save the .mb file to a writable directory (use **`Save As...`** in the **`File`** pull-down menu).

(5) Select **`Create All`** from the **`Build`** pull-down menu of the dialog box. This option creates three files for the module: `averagecell.c`, `averagecell.mdf`, and `averagecell.make`,

(6) Implement the AverageCell function.

Use an editor to add the following lines after “User’s code goes here” in the `averagecell.c` file:

```
int *itemcounts = NULL, i, neighbor;

/* make scratch space to hold the number of items added for each element */
itemcounts = DXAllocate(input_knt*sizeof(int));
if (!itemcounts)
   goto error;

/*
 * first initialize the output data component to zero, and itemcounts to
 * zero.
 */
for (i=0; i<input_knt; i++) {
   output_data[i] = 0;
   itemcounts[i]=0;
}

/* for each data value, add that value to the appropriate items in the
 * output data array. Also increment itemcounts for those cells.
 */
for (i=0; i<input_knt; i++) {
   /* first do itself */
   output_data[i]+=input_data[i];
   itemcounts[i]++;

   /* now do neighbors in fastest-varying dimension */
   neighbor = i-1;
   if (neighbor >= 0 && ((i % (c_counts[1]-1)) != 0)) {
      output_data[neighbor]+=input_data[i];
      itemcounts[neighbor]++;
   }

   neighbor = i+1;
   if (neighbor < input_knt &&(((i+1)%(c_counts[1]-1)) != 0)) {
      output_data[neighbor]+=input_data[i];
      itemcounts[neighbor]++;
   }

   /* now do neighbors in the slowest-varying dimension */
   neighbor = i - (c_counts[1]-1);
   if (neighbor >= 0) {
      output_data[neighbor]+=input_data[i];
      itemcounts[neighbor]++;
   }
   neighbor = i + (c_counts[1]-1);
   if (neighbor < input_knt) {
      output_data[neighbor]+=input_data[i];
      itemcounts[neighbor]++;
   }
}
```

```
   /* now divide by the number of items added for that cell */
   for (i=0; i< input_knt; i++)
      output_data[i] = output_data[i]/itemcounts[i];

   DXFree((Pointer)itemcounts);
   return OK;
error:
   DXFree((Pointer)itemcounts);
   return ERROR;
}
```

The file /usr/lpp/dx/samples/program_guide/averagecell.c contains a completed version of this program.

(7) To create a version of Data Explorer that includes the AverageCell module, enter the command:

```
make -f averagecell.make dxexec
```

(You have now created an executable that contains the AverageCell module.)

(8) To invoke this version, enter:

```
dx -mdf ./averagecell.mdf -exec ./dxexec
```

This command starts Data Explorer (the averagecell.mdf file tells the graphical user interface about AverageCell and its inputs and outputs). The executable dxexec invoked here is the one created in Step 6.

(9) With this version of Data Explorer you can now run any visual program that uses the AverageCell module. One such program is /usr/lpp/dx/samples/program_guide/averagecell.net

Chapter 7. Importing Data

If you want to import data that is not in a format supported by Data Explorer, you have three options:

- Write an import filter to convert the data to Data Explorer or General Array importer format on disk.
- Write an import filter to convert the data to Data Explorer or General Array header format on standard output, and use the external filter option of Import to import the data. (See “Import” on page 165 in *IBM Visualization Data Explorer User’s Reference*.)
- Write your own import module to read the data and create a Data Explorer Object as its output (e.g., a Field Object).

Notes:

1. The following examples illustrate the conversion of data from Hierarchical Data Format (HDF) to Data Explorer file format. Understanding the examples does not require any familiarity with HDF.
2. HDF libraries are not distributed with Data Explorer and no makefiles are provided for the programs used. If you have the HDF libraries, you can use the same compilation and linking procedures as you do for other programs requiring those libraries.
3. The Import module will import HDF data. The example in 7.2, “Writing an Import Module” on page 51 is for illustration only.

7.1 Writing a Filter

The filters used to create a Data Explorer format file on disk and on standard output are essentially the same.

Assume a single data set of scalar data stored in an HDF file. All HDF files are gridded. The dimensionality and size of the grid are to be determined from queries to the data set.

The following C program requires the HDF file name as an argument. It is found in **/usr/lpp/dx/samples/program_guide/simpleimportfilter.c** .

```


#include <stdio.h>
```

df.h is a necessary include file for HDF library routines.

```
#include <df.h>

#define MAXRANK 3

main(argc, argv)
  char *argv[];
{
  FILE *in;
  char filename[80];
  int dims, counts[MAXRANK], numelements, i, j;
  float deltas[MAXRANK*MAXRANK], origins[MAXRANK], *databuf=NULL;

```

Check that the user has supplied the name of the file to be opened.

```
if (argc < 2) {
  fprintf(stderr,"Usage: simpleimportfilter <filename> \n");
  return 0;
}

strcpy(filename, argv[1]);
}
```

The HDF library routine **DFishdf** checks the file for accessibility and for the correct (HDF) format. If the file is not accessible or is not an HDF file, the routine generates an error message.

```
if (DFishdf(filename) != 0) {
    fprintf(stderr,
            "file \"%s\" is not accessible, or is not an hdf file\n",
            filename);
  return 0;
}
```

Initialize the HDF library.

```
DFSDrestart();
```

The HDF library routine **DFSDgetdims** returns the dimensionality of the grid (1D, 2D, etc.) in **dims**. The number of positions in each dimension is returned in the Array **counts**.

```
DFSDgetdims(filename, &dims, counts, MAXRANK);
```

Determine the number of elements in the data Array.

```
numelements=1;
for (i=0; i<dims; i++)  {
   numelements= numelements * counts[i];
}
```

Create a buffer for the data.

```
databuf = (float *)malloc(numelements*sizeof(float));
if (!databuf) {
  fprintf(stderr,"out of memory\n");
  return 0;
}
```

The HDF library routine **DFSDgetdata** reads the data from the HDF file to the data Array.

```
DFSDgetdata(filename, dims, counts, databuf);
```

Write the Data Explorer file format description of the data Array on standard output.

```
printf("object 1 class array type float rank 0 items %d data follows\n",
       numelements);
for (i=0; i<numelements; i++)
   printf(" %f\n ", databuf[i]);
```

Set the dependency of the data to “positions.”

```
printf("attribute \"dep\" string \"positions\"\n ");
```

Now create the position origin and deltas (origin 0 and deltas 1 in each dimension).

```
for (i=0; i<dims; i++) {
   origins[i] = 0.0;
   for (j=0; j<dims; j++) {
     if (i==j)
       deltas[i*dims + j] = 1.0;
     else
       deltas[i*dims + j] = 0.0;
   }
}
```

Write out the connections and positions.

```
switch (dims) {
  case (1):
   printf("object 2 class gridconnections counts %d\n", counts[0]);
   printf("object 3 class gridpositions counts %d\n", counts[0]);
   printf(" origin %f\n", origins[0]);
   printf(" delta  %f\n", deltas[0]);
   break;
  case (2):
   printf("object 2 class gridconnections counts %d %d\n",
           counts[0], counts[1]);
   printf("object 3 class gridpositions counts %d %d\n",
           counts[0], counts[1]);
   printf(" origin %f %f\n", origins[0], origins[1]);
   printf(" delta  %f %f\n", deltas[0], deltas[1]);
   printf(" delta  %f %f\n", deltas[2], deltas[3]);
   break;
  case (3):
   printf("object 2 class gridconnections counts %d %d %d\n",
           counts[0], counts[1], counts[2]);
   printf("object 3 class gridpositions counts %d %d %d\n",
           counts[0], counts[1], counts[2]);
   printf(" origin %f %f %f\n", origins[0], origins[1], origins[2]);
   printf(" delta  %f %f %f\n", deltas[0], deltas[1], deltas[2]);
   printf(" delta  %f %f %f\n", deltas[3], deltas[4], deltas[5]);
   printf(" delta  %f %f %f\n", deltas[6], deltas[7], deltas[8]);
   break;
  default:
   printf(stderr,"dimensionality must be 1D, 2D, or 3D");
   return 0;
}
```

Write out the description of the Field.

```
printf("object 4 class field\n");
printf("  component \"data\" value 1\n");
printf("  component \"connections\" value 2\n");
printf("  component \"positions\" value 3\n");

return 1;

```

7.2 Writing an Import Module

Any external filter, like the one just illustrated, has the disadvantage of running more slowly than an import module because it sends the data to a file or through a socket. A built-in module reads the data into memory, where Data Explorer uses it directly. In this example, the import module SimpleImport reads the same HDF file as the external filter did.

Note: Because the import module is very simple and does not require the traversal of input Fields, the Module Builder is not used in this example. (This C program is also found in **/usr/lpp/dx/samples/program_guide/simpleimportfilter.c**

```
#include <dx/dx.h>
```

df.h is a necessary include file for HDF library routines.

```
#include <df.h>

#define MAXRANK 3

Error m_SimpleImport(Object *in, Object *out)
{
  Array a=NULL;
  Field f=NULL;
  char *filename;
  int dims, counts[MAXRANK], numelements, i, j;
  float deltas[MAXRANK*MAXRANK], origins[MAXRANK], *data;
```

Extract the file name from **in[0]**, and check that it is a string.

```
if (!in[0]) {
  DXSetError(ERROR_BAD_PARAMETER,"missing filename");
  goto error;
}
else if (!DXExtractString(in[0], &filename)) {
  DXSetError(ERROR_BAD_PARAMETER, "filename must be a string");
  goto error;
}
```

The HDF library routine **DFishdf** checks the file for accessibility and for the correct (HDF) format. If the file is not accessible or is not an HDF file, the routine generates an error message.

```
if (DFishdf(filename) != 0) {
   DXSetError(ERROR_BAD_PARAMETER,
              "file \"%s\" is not accessible, or is not an hdf file",
               filename);
   goto error;
}

```

Initialize the HDF library.

```
DFSDrestart();
```

The HDF library routine **DFSDgetdims** returns the dimensionality of the grid (1D, 2D, etc.) in **dims**. The number of positions in each dimension is returned in the Array **counts**.

```
DFSDgetdims(filename, &dims, counts, MAXRANK);
```

Make a new Array (scalar).

```
a = DXNewArray(TYPE_FLOAT, CATEGORY_REAL, 0);
if (!a)
  goto error;
```

Determine the number of elements in the data Array.

```
numelements=1;
for (i=0; i<dims; i++)  {
   numelements= numelements * counts[i];
}
```

Allocate space to the data Array.

```
if (!DXAddArrayData(a, 0, numelements, NULL))
   goto error;
```

Get a pointer to memory for the data Array.

```
data = (float *)DXGetArrayData(a);
if (!data)
   goto error;
```

The HDF library routine **DFSDgetdata** reads the data from the HDF file to the data Array.

```
DFSDgetdata(filename, dims, counts, data);
```

Create a new Field.

```
f = DXNewField();
if (!f)
  goto error;
```

Set the dependency of the data to “positions.”

```
if (!DXSetStringAttribute((Object)a, "dep", "positions"))
  goto error;
```

Set the data Array as the data component of **f**.

```
if (!DXSetComponentValue(f, "data", (Object)a))
  goto error;
a=NULL;
```

Create the connections Array. **DXMakeGridConnections** sets up the element type. Place the connections in the Field.

```
a = DXMakeGridConnectionsV(dims, counts);
if (!a)
   goto error;
if (!DXSetComponentValue(f, "connections", (Object)a))
   goto error;
a=NULL;
```

Now create the position origin and deltas for the position (origin 0 and deltas 1 in each dimension).

```
  for (i=0; i<dims; i++) {
     origins[i] = 0.0;
     for (j=0; j<dims; j++) {
       if (i==j)
         deltas[i*dims + j] = 1.0;
       else
         deltas[i*dims + j] = 0.0;
     }
  }
```

Create the positions Array and place it in the Field.

```

  a = DXMakeGridPositionsV(dims, counts, origins, deltas);
  if (!a)
    goto error;
  if (!DXSetComponentValue(f, "positions", (Object)a))
    goto error;
  a=NULL;
```

DXEndField sets default attributes and creates the bounding box.

```
  if (!DXEndField(f))
    goto error;

```

Set **f** as the first output of the module.

```
  out[0]=f;
  return OK;

```

On error, delete **f** and **a**.

```
 error:
  DXDelete((Object)f);
  DXDelete((Object)a);
  return ERROR;
}
```

Chapter 8. Using the Pick Structure

Data Explorer includes a tool for “picking” points in Objects in an image.

This tool, Pick, creates a structure that can be used to perform various functions (e.g., to display the data value at picked points). But you can also write your own module to perform different functions if you like.

8.1 The Pick Structure

Picking is the selection of a location on an object in an image by use of the mouse. A straight line from the camera through the location selected constitutes a “poke,” which may intersect the object in the image in one or more places or in none at all. The intersections are called “picks.” For example, a poke through a spherical isosurface results in two picks: one on the “front” surface and one on the “back”. Picks differ from probes in that probes may be present anywhere in 3-dimensional space, picks only on the surface of an object.

The pick structure is a Field, and the picked points are listed in its “positions” component. A number of routines in Data Explorer allow you to query the pick structure output by the Pick tool and to traverse a picked Object. (See 13.6, “Pick-Assistance Routines” on page 142 for details.) The structure includes information on how to traverse the picked Object to reach the picked element. It also identifies:

- the connection in which the picked point resides (the element ID)
- the vertex of the picked element closest to the picked point (the vertex ID)
- the position of the picked point itself.

If the picked Object has no connections, the element ID and the vertex ID both refer to the position closest to the picked point. Other information can be accessed with pick-assistance routines.

Note: For a module that uses the pick structure, the Object displayed in the image being picked must (1) be the same as the Object passed to the module or (2) have a matching Object hierarchy. The reason for this requirement is that the output of the Pick tool describes the location of the picked Object as it exists in the hierarchy of the rendered Object. To use the pick structure, therefore, requires an Object with a matching structure.

8.2 ShowPick Module Example—Using Color to Show a Picked Object

In the following example, the ShowPick module colors the entire object in white, except for the Field, element, or vertex containing the pick point(s). The color of the latter is specified by the user.

The module description file for ShowPick is:

```
MODULE ShowPick
CATEGORY User
DESCRIPTION sets a triangle in a picked Field to a particular color
INPUT input; object; (none); object with picked points
INPUT pickobject; field; (none); picking structure
INPUT color; string; "red"; color to set
INPUT colorwhich; integer; 0; color the element (0), vertex (1) or entire field (2)
INPUT poke; integer; (all); poke selection
INPUT pick; integer; (all); pick selection
INPUT depth; integer; (bottom); selection depth
OUTPUT output; object; object with picked structures marked using color
```

As the **.mdf** file shows, the ShowPick module takes seven inputs and generates one output. To create a version of Data Explorer that includes this module, copy the following files to the directory where you want to work:

/usr/lpp/dx/samples/program_guide/Makefile_ *supported workstation model*
/usr/lpp/dx/samples/program_guide/showpick.c
/usr/lpp/dx/samples/program_guide/showpick.mdf

Now rename the makefile to **Makefile** and enter: `make showpick`. This command creates an executable that contains the ShowPick module.

To invoke this version (from the directory to which the files were copied), enter:

```
dx -mdf ./showpick.mdf -exec ./dxexec
```

This command starts Data Explorer (the **showpick.mdf** file tells the graphical user interface about ShowPick and its inputs and outputs). With this version of Data Explorer you can now run any visual program that uses the ShowPick module. One such program is **showpick.net** in the **/usr/lpp/dx/samples/program_guide** directory.

```
#include <dx/dx.h>
#include "pick.h"

static Error DoPick(Object, Object, RGBColor, int, int, int, int);
static Error SetColor(Object, RGBColor);

Error m_ShowPick(Object *in, Object *out)
{
  Object o = NULL, pickfield;
  char *colorstring;
  int colorwhich, poke, pick, depth;
  RGBColor color;
```

Copy the structure of **in[0]**, the object in which picking took place.

```
  if (!in[0]) {
    DXSetError(ERROR_BAD_PARAMETER, "missing input");
    goto error;
  }
  o = (Object)DXCopy(in[0], COPY_STRUCTURE);
  if (!o)
    goto error;
```

First, set all the colors to white, to initialize. (The SetColor routine is defined below.)

```
  if (!SetColor(o, DXRGB(1.0, 1.0, 1.0)))
    goto error;
```

in[1] is the pick Field. If the pick Field is **NULL** or an empty Field, just return the copy of the object.

```
  if (!in[1] || DXEmptyField(in[1])) {
    out[0] = o;
    return OK;
  }
  pickfield = in[1];
```

Get the color that will be used for picked Objects, which is **in[2]**.

```
  if (in[2]) {
    if (!DXExtractString((Object)in[2], &colorstring)) {
      DXSetError(ERROR_BAD_PARAMETER,"color must be a string");
      goto error;
    }
```

Convert the color name to an RGB vector.

```

    if (!DXColorNameToRGB(colorstring, &color))
      goto error;
  }
  else {
```

If **in[2]** is not specified, then the default color is red.

```
    color = DXRGB(1.0, 0.0, 0.0);
  }
```

Determine if we are to color just the picked element, just the vertex closest to the picked point, or the entire Field. The default is to color just the picked element.

```
  if (!in[3]) {
    colorwhich = 0;
  }
  else {
    if (!DXExtractInteger(in[3], &colorwhich)) {
      DXSetError(ERROR_BAD_PARAMETER,"colorwhich flag must be 0, 1, or 2");
      goto error;
    }
    if ((colorwhich < 0)&&(colorwhich > 2)) {
      DXSetError(ERROR_BAD_PARAMETER,"colorwhich flag must be 0, 1, or 2");
      goto error;
    }
  }
```

Determine if we are to select a particular poke, or all of them. The default is to select all of them.

```

    if (!in[4]) {
      poke = -1;
    }
    else {
      if (!DXExtractInteger(in[4], &poke)) {
        DXSetError(ERROR_BAD_PARAMETER,"poke must be a nonnegative integer");
        goto error;
      }
      if (poke < 0) {
        DXSetError(ERROR_BAD_PARAMETER,"poke must be a nonnegative integer");
        goto error;
      }
    }
```

Determine if we are to select a particular pick, or all of them. The default is to select all of them.

```
    if (!in[5]) {
      pick = -1;
    }
    else {
      if (!DXExtractInteger(in[5], &pick)) {
        DXSetError(ERROR_BAD_PARAMETER,"pick must be a nonnegative integer");
        goto error;
      }
      if (pick < 0) {
        DXSetError(ERROR_BAD_PARAMETER,"pick must be a nonnegative integer");
        goto error;
      }
    }
```

Determine if we are to select a depth. The default is to select the deepest level.

```
    if (!in[6]) {
      depth = -1;
    }
    else {
      if (!DXExtractInteger(in[6], &depth)) {
        DXSetError(ERROR_BAD_PARAMETER,"depth must be a nonnegative integer");
        goto error;
      }
      if (depth < 0) {
        DXSetError(ERROR_BAD_PARAMETER,"depth must be a nonnegative integer");
        goto error;
      }
    }
```

Traverse the picked object, using the pick structure, passing the given parameters.

```
    if (!DoPick(o, pickfield, color, colorwhich, poke, pick, depth))
      goto error;
```

Delete the **opacities** component.

```
    if (DXExists(o, "opacities"))
      DXRemove(o,"opacities");
```

Successful return.

```
  out[0] = o;
  return OK;
```

Return on error.

```
 error:
  DXDelete(o);
  return ERROR;
}
```

The **DoPick()** routine traverses the picked object.

```
static
  Error
  DoPick(Object o, Object pickfield, RGBColor color, int colorwhich,
     int pokes, int picks, int depth)
{
  int pokecount, pickcount, poke, pick, i, pathlength;
  int vertexid, elementid, *path, numitems, index;
  Object current;
  Matrix matrix;
  Array a, newcolors=NULL, oldcolors;
  char *depatt;
  RGBColor *newcolors_ptr, oldcolor;
  int pokemin, pokemax;
  int pickmin, pickmax;
  int thisdepth;
```

pickfield is expected to be a Field.

```
  if (!(DXGetObjectClass(pickfield)==CLASS_FIELD)) {
    DXSetError(ERROR_INVALID_DATA,"pickfield must be a field");
    goto error;
  }
```

Find out the number of pokes.

```
  DXQueryPokeCount(pickfield, &pokecount);
```

The user has chosen to mark all pokes.

```
  if (pokes < 0) {
    pokemin = 0, pokemax = pokecount-1;
  }
```

The user has specified a poke larger than the number present.

```
  else if (pokes > pokecount-1) {
    DXSetError(ERROR_BAD_PARAMETER,
           "only %d pokes are present", pokecount);
    return ERROR;
  }
```

Consider only the specified poke.

```
  else
    pokemin = pokemax = pokes;
```

For each poke...

```
  for (poke=pokemin; poke<=pokemax; poke++) {
```

Find out how many picks there are in this poke.

```
    if (!DXQueryPickCount(pickfield, poke, &pickcount))
      goto error;
```

Issue warning if this particular poke does not contain as many picks as the user has specified.

```
    if (picks > pickcount-1) {
      DXWarning("poke %d contains only %d picks", poke, pickcount);
    }

    else {
      if (picks < 0) {
        pickmin = 0, pickmax = pickcount-1;
      }
      else {
        pickmin = pickmax = picks;
      }
```

For each pick...

```

      for (pick=pickmin; pick<=pickmax; pick++) {
```

For the given **pickfield**, the current poke number, and the current pick number, get the traversal path **path**, the length of the traversal path **pathlength**, and the IDs of the picked element and the picked vertex.

```
DXQueryPickPath(pickfield, poke, pick, &pathlength, &path,
                &elementid, &vertexid);
```

Initialize **current** to the picked object, and **matrix** to the identity matrix.

```
current = o;
matrix = Identity;
if (depth != -1 && pathlength > depth)
  thisdepth = depth;
else
  thisdepth = pathlength;
```

Iterate through the pick path.

```
for (i=0; i<thisdepth; i++) {
  current = DXTraversePickPath(current, path[i], &matrix);
  if (!current)
    goto error;
}
```

current is now the Field level of the picked Object, and we have the element and vertex IDs of the picked object.

```
if (colorwhich == 2 || DXGetObjectClass(current) != CLASS_FIELD) {
```

We are simply to color the entire Field.

```
  if (!SetColor(current, color))
    goto error;
}
else {
```

Otherwise, we want to set the indicated element or vertex to the given color. We start by making a new colors component (not compact), but only if the input colors component is still compact. If it is already expanded, then modify it.

First, determine the dependency of the colors.

```
if (colorwhich == 0) {
     if (a = DXGetComponentValue(current, "connections")) {
         index = elementid;
         depatt = "connections";
     }
     else if (a = DXGetComponentValue(current, "faces")) {
         index = elementid;
         depatt = "faces";
     }
     else {
          a = DXGetComponentValue(current, "positions");
          index = vertexid;
          depatt = "positions";
     }
}
else {
  a = DXGetComponentValue(current, "positions");
  index = vertexid;
  depatt = "positions";
}
```

Determine the number of items.

```
if (!DXGetArrayInfo(a, &numitems,NULL,NULL,NULL,NULL))
  goto error;
```

If the traversal index is greater than the number of items, something is wrong.

```
   if (index >= numitems) {
     DXSetError(ERROR_INVALID_DATA,
             "pick structure does not correspond to picked object");
     goto error;
   }
```

Get the original colors component.

```
   oldcolors = DXGetComponentValue((Field)current, "colors");
```

If it is a constant Array, we need to expand it so that we can set just one element or vertex to the given color.

```
if (DXQueryConstantArray(oldcolors, NULL, &oldcolor)) {
```

Create a new colors Array and allocate space to it.

```
  newcolors = DXNewArray(TYPE_FLOAT,CATEGORY_REAL, 1, 3);
  if (!DXAddArrayData(newcolors, 0, numitems, NULL))
    goto error;
```

Start by setting all colors to the original constant color.

```
  newcolors_ptr = (RGBColor *)DXGetArrayData(newcolors);
  for (i=0; i<numitems; i++) {
    newcolors_ptr[i] = oldcolor;
  }
```

Replace the colors in the Field with the new colors component.

```
      if (!DXSetComponentValue((Field)current, "colors",
                       (Object)newcolors))
        goto error;
      newcolors=NULL;

      DXSetComponentAttribute((Field)current, "colors", "dep",
                       (Object)DXNewString(depatt));
    }


    else {
```

The colors are already expanded, presumably from an earlier pick in this Field.

```
      newcolors_ptr = (RGBColor *)DXGetArrayData(oldcolors);
    }
```

Set the correct triangle or position to the given color.

```
    newcolors_ptr[index] = color;
   }
  }
 }
}

 return OK;

  error:
   DXDelete((Object)newcolors);
   return ERROR;
 }
```

This routine sets all colors in object **o** to the given color.

```
 static Error SetColor(Object o, RGBColor color)
 {
   Object subo;
   Array a, newcolors=NULL;
   int numitems, i;


   switch (DXGetObjectClass(o)) {


   case (CLASS_GROUP):

```

If **o** is a Group, call **SetColor** recursively on its children.

```
     for (i=0; subo = DXGetEnumeratedMember((Group)o, i, NULL); i++))
       SetColor(subo, color);
     break;


   case (CLASS_XFORM):
```

If **o** is an Xform, call **SetColor** on its child.

```
     DXGetXformInfo((Xform)o, &subo, NULL);
     SetColor(subo, color);
     break;


    case (CLASS_CLIPPED):
```

If **o** is a Clipped object, call **SetColor** on its child.

```
     DXGetClippedInfo((Clipped)o, &subo, NULL);
     SetColor(subo, color);
     break;


    case (CLASS_FIELD):
```

If **o** is a Field, set the colors to the given color.

```
     if (DXEmptyField((Field)o))
      return OK;
```

The number of colors and the dependency of the colors will depend on whether connections are present. If not, it is checked for the presence of faces. Otherwise, the colors will be dependent on positions.

```
     if (a = DXGetComponentValue((Field)o, "connections")) {
      DXGetArrayInfo(a, &numitems, NULL, NULL, NULL, NULL);
      newcolors = (Array)DXNewConstantArray(numitems, &color,
                                TYPE_FLOAT,
                                CATEGORY_REAL, 1, 3);
      DXSetComponentValue((Field)o, "colors", (Object)newcolors);
      newcolors = NULL;
      DXSetComponentAttribute((Field)o,"colors", "dep",
                        (Object)DXNewString("connections"));
     }
     else if (a = DXGetComponentValue((Field)o, "faces")) {
      DXGetArrayInfo(a, &numitems, NULL, NULL, NULL, NULL);
      newcolors = (Array)DXNewConstantArray(numitems, &color,
                                TYPE_FLOAT,
                                CATEGORY_REAL, 1, 3);
      DXSetComponentValue((Field)o, "colors", (Object)newcolors);
      newcolors = NULL;
      DXSetComponentAttribute((Field)o,"colors", "dep",
                        (Object)DXNewString("faces"));
     }
```

```
    else {
      a = DXGetComponentValue((Field)o, "positions");
      DXGetArrayInfo(a, &numitems, NULL, NULL, NULL, NULL);
      newcolors = (Array)DXNewConstantArray(numitems, &color,
                              TYPE_FLOAT,
                              CATEGORY_REAL, 1, 3);
      DXSetComponentValue((Field)o, "colors", (Object)newcolors);
      newcolors = NULL;
      DXSetComponentAttribute((Field)o,"colors", "dep",
                      (Object)DXNewString("positions"));
    }

    break;
  }

```

Successful return or return on error.

```

  return OK;
 error:
  DXDelete((Object)newcolors);
  return ERROR;
}
```

Chapter 9. Writing Modules for a Parallel Environment

Writing a “parallel” module involves considerations beyond those encountered in using the Module Builder.

9.1 A Parallel Version of the Add Module

The Add module created in the example in 4.1, “Add Module Example—Add a Number to Every Data Value” on page 32 would work correctly on partitioned data because the code generated by the Module Builder automatically provides recursive traversal. However, it would not run in parallel on a parallel-architecture machine. To create an “addparallel” module, copy the following files to the directory you want to work in:

/usr/lpp/dx/samples/program_guide/Makefile_*workstation*
/usr/lpp/dx/samples/program_guide/add_parallel.c
/usr/lpp/dx/samples/program_guide/addpar.mdf

Now rename the makefile to **Makefile** and enter `make add_par`.

To run this module in Data Explorer (from the directory to which the files were copied), enter:

```
dx -mdf ./addpar.mdf -exec ./dxexec
```

This command starts Data Explorer (the **addpar.mdf** file tells the graphical user interface about AddParallel and its inputs and outputs).

You can now run any visual program that uses the AddParallel module. One such program is **/usr/lpp/dx/samples/program_guide/add_parallel.net**.

The AddParallel module:

- Encapsulates the Field-level processing in the subroutine **task** in this example.
- Calls **DXCreateTaskGroup** just before recursively traversing the Object in **m_AddParallel**.
- Adds the tasks for processing the Fields during recursive traversal by calling **DXAddTask**.
- Calls **DXExecuteTaskGroup** just after recursive traversal. At this point, the tasks that are defined will be scheduled on multiple processors. If any of the tasks returns an error, that error will be returned from **DXExecuteTaskGroup**.

```
#include <dx/dx.h>

static Error DoAdd(Object o, float x);

m_AddParallel(Object *in, Object *out)
{
    Object o = NULL;
    float x;
```

Copy the structure of **in[0]**.

```
    if (!in[0])
        DXErrorGoto(ERROR_BAD_PARAMETER, "missing object");
    o = DXCopy(in[0], COPY_STRUCTURE);
    if (!o)
        goto error;
```

Extract floating-point parameter from **in[1]** (default 0).

```
   if (!in[1])
       x = 0;
   else if (!DXExtractFloat(in[1], &x))
       DXErrorGoto(ERROR_BAD_PARAMETER, "bad addend");
```

Create the task Group, call **DoAdd()** for recursive traversal, and then execute the task Group.

```
    DXCreateTaskGroup();
    if (!DoAdd(o, x)) {
      DXAbortTaskGroup()
      goto error;
    }
    if (!DXExecuteTaskGroup())
      goto error;
```

A successful return or return on error.

```
    out[0] = o;
    return OK;

error:
    DXDelete(o);
    return ERROR;
}


```

The argument block for passing parameters to the task routine:

```
struct arg {
    Field field;
    float x;
}
```

The following task routine does the actual work of processing a Field. **DXAddTask** instructs the executive to call this routine once for each Field. The executive will pass to **task** the argument block pointer that was specified when **DXAddTask** itself was called.

```

static Error
task(Pointer p)
{
    struct arg *arg = (struct arg *)p;
    Field field;
    float x, *from, *to;
    int i, n;
    Array a;
```

Extract the arguments.

```
    field = arg->field;
    x = arg->x;
```

Extract, typecheck, and get the data from the “data” component.

```
    a = (Array) DXGetComponentValue(field, "data");
    if (!a)
        DXErrorReturn(ERROR_MISSING_DATA, "field has no data");
    if (!DXTypeCheck(a, TYPE_FLOAT, CATEGORY_REAL, 0))
        DXErrorReturn(ERROR_BAD_TYPE, "data is not floating point");
    from = (float *) DXGetArrayData(a);
```

Create a new Array, allocate space to it, and put it in the Field.

```
    DXGetArrayInfo(a, &n, NULL, NULL, NULL, NULL);
    a = DXNewArray(TYPE_FLOAT, CATEGORY_REAL, 0);
    if (!DXAddArrayData(a, 0, n, NULL))
        return ERROR;
    to = (float *) DXGetArrayData(a);
    DXSetComponentValue(field, "data", (Object)a);
```

The following loop adds **x** to obtain the result.

```
    for (i=0; i<n; i++)
        to[i] = from[i] + x;
```

Clean up the Field.

```
        DXChangedComponentValues(field, "data");
        DXEndField(field);

        return OK;
    }
```

The recursive traversal routine follows. Note that at this point (and for each Field) it does not process the Field but calls **DXAddTask**, specifying the routine that will be called in parallel to do the actual work.

The Data Explorer programming interface is designed so that, in general, the programmer does need to use explicit locks. For information about local and global memory allocation, see 12.3, “Memory Allocation” on page 116.

```
static
Error
DoAdd(Object o, float x)
{
    struct arg arg;
    int i, n;
    Object oo;
```

Determine the class of the object.

```
    switch (DXGetObjectClass(o)) {

    case CLASS_FIELD:
```

Add the task for this Field.

```
        arg.field = (Field)o;
        arg.x = x;
        if (!DXAddTask(task, &arg, sizeof arg, 0.0))
            return ERROR;
        break;

    case CLASS_GROUP:
```

Traverse Groups recursively.

```
        for (i=0; oo=DXGetEnumeratedMember((Group)o, i, NULL); i++)
            if (!DoAdd(oo, x))
                return ERROR;
        break;
    }

    return OK;
}
```

9.2 A Parallel Version of the AverageCell Module

Writing a version of AverageCell for a parallel environment introduces a “problem” that does not arise with the Add module: the implementation of parallelism by dividing Fields into spatially disjoint subsets called partitions. Each partition is stored as a Field inside a Group Object. This Group is a special subclass of Group objects called a “Composite Field.”

The AverageCell algorithm requires information about the neighbors of each cell. But for cells on a partition boundary, at least some of those neighbors are in another partition. **DXGrow** deals with this difficulty and obtains the needed information by “growing” the partition by a specified number of cells. In effect it “restores the old neighborhood.” The desired operation can then be performed on the “grown” Field. **DXShrink** restores the partition to its pre-growth state by removing the extra cells and “cleaning up.” (See 13.4, “Growing and Shrinking Partitioned Data” on page 137.)

To create a version of Data Explorer that includes the AverageCellParallel module, copy the following files to the directory where you want to work:

/usr/lpp/dx/samples/program_guide/Makefile_*workstation*
/usr/lpp/dx/samples/program_guide/averagecell_parallel.c
/usr/lpp/dx/samples/program_guide/averagecellpar.mdf

Now rename the makefile to **Makefile** and enter: `make avgcell_par`.

To run this version (from the directory to which the files were copied), enter:

```
dx -mdf ./averagecellpar.mdf -exec ./dxexec
```

You can now run any visual program that uses the AverageCellParallel module. One such program is **averagecell_parallel.net** in the directory **/usr/lpp/dx/samples/program_guide**.

The example AverageCellParallel code follows:

```
#include <dx/dx.h>

static Error DoAverageCell(Object);



Error m_AverageCellParallel(Object *in, Object *out)
{
  Object o=NULL;

  if (!in[0]) {
    DXSetError(ERROR_BAD_PARAMETER,"missing input");
    goto error;
  }

  o = DXCopy(in[0], COPY_STRUCTURE);
```

“Grow” the Fields so that averaging can be performed across partition boundaries. Since it is not necessary to grow a Field beyond the original boundaries of the data, and since only the “data” component is affected, grow the partition by one cell. (The original components—“positions,” “data,” etc.—are copied into components named “original positions,” “original data,” and so on.)

```
  if (!DXGrow(o, 1, GROW_NONE, "data", NULL))
    goto error;
```

Create the task Group.

```
  if (!DXCreateTaskGroup())
    goto error;
```

The add tasks will be added in **DoAverageCell()**.

```
  if (!DoAverageCell(o)) {
    DXAbortTaskGroup();
    goto error;
  }

  if (!DXExecuteTaskGroup())
    goto error;
```

Do not call **DXShrink** to shrink the grown Field until you have recursively removed any “original data” component(s), assuming that you want to save the newly created one(s). Otherwise the new “data” component(s) will be replaced by the (unprocessed) “original data” components(s). Now you can call **DXShrink**.

```
  if (DXExists(o, "original data"))
    DXRemove(o,"original data");
  if (!DXShrink(o))
    goto error;

  out[0] = o;
  return OK;
 error:
  DXDelete((Object)o);
  return ERROR;
}

struct arg {
  Field field;
};

static Error AddCellTask(Pointer p)
{
  struct arg *arg = (struct arg *)p;
  int i, j, numitems, shape, *neighbors_ptr, sum, neighbor;
  int dim, counts[3];
  char *attribute;
  float *data_ptr, *newdata_ptr, dataaverage;
  Array connections, data, newdata=NULL, neighbors;
  Field field;

  field = arg->field;

```

Get the connections component; determine the number of connections and their element type.

```

  connections = (Array)DXGetComponentValue(field,"connections");
  if (!connections) {
    DXSetError(ERROR_MISSING_DATA,"input has no connections");
    goto error;
  }
  if (!DXGetArrayInfo(connections, &numitems, NULL, NULL, NULL, NULL)) {
    goto error;
  }
  if (!(attribute=
        (char *)DXGetString((String)DXGetComponentAttribute(field,
                                                             "connections",
                                                             "element type")))) {
    DXSetError(ERROR_MISSING_DATA,
               "missing connection element type attribute");
    goto error;
  }


```

Get the data component, and get the data dependency attribute.

```
data = (Array)DXGetComponentValue(field,"data");
if (!data) {
  DXSetError(ERROR_MISSING_DATA,"input has no data");
  goto error;
}
if (!(attribute=
      (char *)DXGetString((String)DXGetComponentAttribute(field,
                                                           "data",
                                                           "dep")))) {
  DXSetError(ERROR_MISSING_DATA,
             "missing data dependency attribute");
  goto error;
}
```

In this example, the data must be dependent on the connections.

```
if (strcmp(attribute,"connections")) {
  DXSetError(ERROR_INVALID_DATA,
             "data must be dependent on connections");
  goto error;
}
```

For this example, the data must be floating-point scalar.

```
if (!DXTypeCheck(data, TYPE_FLOAT, CATEGORY_REAL, 0, NULL)) {
  DXSetError(ERROR_INVALID_DATA, "data must be floating point scalar");
  goto error;
}
```

Get a pointer to the data.

```
data_ptr = (float *)DXGetArrayData(data);
```

Make a new data component, allocate space in it, and get a pointer to it.

```
newdata = DXNewArray(TYPE_FLOAT,CATEGORY_REAL, 0);
if (!DXAddArrayData(newdata, 0, numitems, NULL))
  goto error;
newdata_ptr = (float *)DXGetArrayData(newdata);
```

If the data is ungridded, use the neighbors component. If it is gridded, use a different method.

```
if (!DXQueryGridConnections(connections, &dim,  counts)) {
```

Now the program needs the neighbors of the connections. Note that neighbors can be obtained only for ungridded data: for gridded data there are more efficient ways to determine neighbors.

```
    neighbors = DXNeighbors(field);
    if (!neighbors)
      goto error;
    neighbors_ptr = (int *)DXGetArrayData(neighbors);
    if (!DXGetArrayInfo(neighbors, NULL, NULL, NULL, NULL, &shape))
      goto error;


    for (i=0; i<numitems; i++) {
      dataaverage = data_ptr[i];
      sum = 1;
```

shape is the number of neighbors of a connection element.

```
      for (j=0; j<shape; j++) {
        neighbor = neighbors_ptr[shape*i + j];
        if (neighbor != -1) {
          dataaverage = dataaverage + data_ptr[neighbor];
          sum++;
        }
      }
      dataaverage = dataaverage/sum;
      newdata_ptr[i] = dataaverage;
    }
  }

  else {
```

The connections are gridded. This example handles only 2-dimensional connections (quads).

```

    if (dim != 2) {
      DXSetError(ERROR_INVALID_DATA,"connections must be 2-dimensional");
      goto error;
    }

    for (i=0; i< numitems; i++) {
      dataaverage = data_ptr[i];
      sum = 1;
```

There are as many as four (4) neighbors for every quad.

```
      if ((i % (counts[1]-1)) > 0) {
        neighbor = i-1;
        dataaverage = dataaverage + data_ptr[neighbor];
        sum++;
      }
      if ((i % (counts[1]-1)) < (counts[1] - 2)) {
        neighbor = i+1;
        dataaverage = dataaverage + data_ptr[neighbor];
        sum++;
      }
```

```
151        neighbor = i-(counts[1]-1);
152        if (neighbor>=0 && neighbor<numitems) {
153          dataaverage = dataaverage + data_ptr[neighbor];
154          sum++;
155        }
156        neighbor = i+(counts[1]-1);
157        if (neighbor>=0 && neighbor<numitems) {
158          dataaverage = dataaverage + data_ptr[neighbor];
159          sum++;
160        }
161        dataaverage = dataaverage/sum;
162        newdata_ptr[i] = dataaverage;
163      }
164    }
```

Place the new data component in the Field.

```
165    DXSetComponentValue(field, "data", (Object)newdata);
166    newdata=NULL;
```

The data component has been changed (lines 162 and 165)

```
167    if (!DXChangedComponentValues(field,"data"))
168      goto error;
169
170
171    return OK;
172   error:
173    DXDelete((Object)newdata);
174    return ERROR;
175  }
176
177
178  static Error DoAverageCell(Object object)
179  {
180    Object subo;
181    struct arg arg;
182    int i;
183
184    switch (DXGetObjectClass(object)) {
185    case (CLASS_FIELD):
186
187      arg.field = (Field)object;
188      if (!DXAddTask(AddCellTask, &arg, sizeof(arg), 0.0))
189        goto error;
190      break;
191
192    case (CLASS_GROUP):
```

If **object** is a Group, recursively call **DoAverageCell()**.

```
      for (i=0; subo=DXGetEnumeratedMember((Group)object, i, NULL); i++) {
        if (!DoAverageCell(subo))
          return ERROR;
      }
      break;
    }
    return OK;
  error:
    return ERROR;
}
```

Chapter 10. Making a Module Work

This chapter discusses module description files and the compiling, linking, and debugging of modules.

10.1 Module Description Files

A module description file (**.mdf** file) contains essential information about Data Explorer modules, including their inputs and outputs. Data Explorer uses this information for various executive and user-interface operations, among them the creation of tool icons.

A module description file consists of one or more "definition" sections, one section for each module described. Every section must contain the first two statements shown here, along with **INPUT** and **OUTPUT**:

```
MODULE name
CATEGORY category name
DESCRIPTION module description
FLAGS optional flags
OUTBOARD "executable"; host
LOADABLE "executable"
INPUT name [visible]; type; default; description
OPTIONS option1; option2;...;
OUTPUT name [cache]; type; description
REPEAT n
```

Note: A module description may contain an **OUTBOARD** or a **LOADABLE** statement, but not both.

MODULE Is required and must be the first statement in the definition section. It assigns a name to the module being described.

name must be a single alphanumeric word, with a letter for the first character.

CATEGORY Is required. It assigns the module to a Data Explorer or user-defined category. (Categories function as tool menus in the VPE window; see Chapter 6, "Graphical User Interface: Important Windows" on page 73 in *IBM Visualization Data Explorer User's Guide*.)

category name may contain more than one word (e.g., "Import and Export").

DESCRIPTION Is optional. It serves as a help function.

module description should briefly describe the module function. Brevity is recommended since this description shares limited space with other information (accessed with the **Description...** button in the module's configuration dialog box).

FLAGS Is optional. Most modules do not need to set flags.

- **PIN**: Specifies that a module is always to execute on the same processor. Applicable only to multiprocessor systems.
- **PERSISTENT**: Specifies that the outboard executable is not to be terminated after each execution of the visual program.
- **ERR_CONT**: Specifies that modules downstream are to continue to execute even when this module returns **ERROR**.

- **SIDE_EFFECT**: Specifies that the module has side effects and must execute each time the visual program is executed, even if its inputs have not changed.
- **ASYNC**: Identifies the module as being able to initiate execution in response to an external event. (See also 10.2, "Asynchronous Modules" on page 84.)

OUTBOARD Is optional. It identifies the module as a separate executable program.

Note: If this statement is included, the module definition must not have a **LOADABLE** statement (see below).

"*executable*" specifies the name of the executable and any arguments to be passed. (Quotation marks are required for executable specifications containing spaces or tabs; otherwise they are optional.)

Note: If you are running Data Explorer on the IBM POWER Visualization System**, the name of the executable must be preceded by the term "os," and the combination enclosed in quotation marks (e.g., "os executable").

host is optional and specifies a remote machine on which the executable is to be run. The default host is the one on which the executive runs. (See also "...as an outboard module" on page 8 and 10.5, "Compiling, Linking, and Debugging an Outboard Module" on page 86.)

LOADABLE Is optional. It identifies the module as being runtime loadable (i.e., compiled separately and loaded into Data Explorer at run time.

Note: If this statement is included, the module definition must not have an **OUTBOARD** statement (see above).

"*executable*" specifies the name of the executable and any arguments to be passed. (Quotation marks are required for executable specifications containing spaces or tabs; otherwise they are optional.)

See also "...as a runtime-loadable module" on page 8 and 10.6, "Compiling, Linking, and Debugging a Runtime-loadable Module" on page 91

INPUT Is required for each input parameter (i.e., two input parameters, two statements). A statement consists of four fields separated by semicolons:

1. *name* (of a parameter) must be one word and must conform to the executive's lexical conventions (see Chapter 10, "Data Explorer Scripting Language" on page 187 in *IBM Visualization Data Explorer User's Guide*).

   [*visible*] is optional. **visible:***n* specifies the accessibility and initial visibility of input tabs:
   0: Not initially visible.
   1: Initially visible (default).
   2: Not available to the user interface.

A hidden parameter (**visible:0**) can be exposed with the **Expand** button in the module's configuration dialog box. Less commonly used parameters are often hidden by default.

2. *type* specifies the type(s) of the input and is used for type matching in the Visual Program Editor. The valid types are:

camera	**integer list**	**scalar**	**value**
field	**matrix**	**scalar list**	**value list**
flag	**matrix list**	**series**	**vector**
group	**object**	**string**	**vector list**
integer			

   To specify more than one type, use the word **or** as a separator (see, for example, the description file for Filter in "Examples of Module Description Files" on page 83).

   If the type of the input value is not explicit (e.g., a string without quotation marks or a vector without brackets), the user interface attempts to match the input against the type(s) specified in the **INPUT** statement. It reads from left to right and stops at the first successful match. For this reason, **string** should be specified last, because any series of characters can always be converted to a string by adding double-quotation marks.

3. *default* identifies the value to be used if none has been specified.

   Note: This part of the **INPUT** statement is informational only: it is the module writer's responsibility to implement a default value.

   By convention, parentheses identify a description of default behavior rather than an actual value. If no default is applicable, specify **(no default)**. If the parameter is required, specify **(none)**.

4. *description* should contain a short phrase describing the parameter.

OPTIONS Is optional. It identifies a list of possible values for the parameter. This list can be accessed by clicking on the **...** button to the right of the **Value** field in the module's configuration dialog box.

Options in the list are separated by a semicolon (;). If the option itself includes a semicolon, use a back slash (\) to escape it with. To accommodate inputs that have more options than will fit on a single line, use multiple OPTIONS statements. If the REPEAT statement is used, the OPTIONS statement must precede it.

OUTPUT Is required for each output parameter (i.e., two output parameters, two statements). A statement consists of three fields separated by semicolons:

1. *name* (of a parameter) must be one word and must conform to the executive's lexical conventions (see Chapter 10, "Data Explorer Scripting Language" on page 187 in *IBM Visualization Data Explorer User's Guide*).

   [*attribute*] is optional. **cache:***n* specifies the caching to be performed by the executive:
   0: Do not cache the output.
   1: Cache all outputs (default).
   2: Cache the output of the last execution only.

Output caching is similar to module caching (see “Function Call Attributes” on page 202 in *IBM Visualization Data Explorer User’s Guide*.) Cache specifications for outputs override those for the module.

2. *type* specifies the type of the output and is used for type matching in the Visual Program Editor. The valid types are:

camera	**integer list**	**scalar**	**value**
field	**matrix**	**scalar list**	**value list**
flag	**matrix list**	**series**	**vector**
group	**object**	**string**	**vector list**
integer			

   To specify more than one type, use the word **or** as a separator.

3. *description* should be a short phrase describing the parameter.

REPEAT Is optional. It specifies some number of **INPUT** or **OUTPUT** statements to be repeated. The parameter *n* specifies the number of statements (input or output) affected: “1” specifies the first immediately preceding statement; “2”, the first and second preceding statements; and so on.

REPEAT must come immediately after **INPUT** (after the last input statement if there are two or more) or after **OPTIONS** if **OPTIONS** is used. The same requirement applies to **OUTPUT**. That is, one **REPEAT** for all inputs and another for all outputs.

The number of repetitions of a single statement is determined by the number of corresponding tabs on the module icon (up to a maximum of 21). Thus, **REPEAT** makes it possible to add input and output tabs to (or delete them from) a module icon, thereby adding or deleting inputs and outputs.

Examples of Module Description Files

The following examples illustrate the specification of three modules: Filter, Options, and ShowBox.

The module description for Filter is:

```
MODULE Filter
CATEGORY Transformation
DESCRIPTION  applies a filter to a field
INPUT input;  field;  (none);  data to filter
INPUT filter;  value or string;  "gaussian";  filter to use
INPUT component[visible:0];  string;  "data";  component to be operated on
OPTIONS data; colors
INPUT mask[visible:0]; value or string; “box”; rank-value filter max
OUTPUT output;  field;  filtered data
```

The Filter module is assigned to the Transformation category. It takes four inputs:

Module Input	Type	Default	Description
input	field	none	data to be filtered
filter	value or string	"gaussian"	filter to be used
component	string	"data"	component to be operated on

Module Input	Type	Default	Description
mask	value or string	"box"	rank-value filter maximum

All input parameters but **input** are assigned default values. The **component** and **mask** parameters are hidden by default ([visible:0]).

The OPTIONS line in the module description specifies possible values for the **component** parameter (two in this case). This list of values can be accessed by clicking on the **...** button to the right of the **Value** field in the module's configuration dialog box.

The module description for the ShowBox module is:

```
MODULE ShowBox
CATEGORY Realization
DESCRIPTION  draws a bounding box
INPUT input;  field;  (none);  the field of which to show the bounding box
OUTPUT box;  field;  renderable bounding box of input field
OUTPUT center; vector;  center of bounding box
```

The ShowBox module is assigned to the Realization category. It takes an input, named **input**, of type **field**. There are two outputs, named **box** and **center**, of type **field** and **vector** respectively.

The module description for the Options module is:

```
MODULE Options
CATEGORY Structuring
DESCRIPTION  associates attributes with an object
INPUT input;  object;  (none); object with attributes to be set
INPUT attribute;  string;  (no default);  attribute to set
INPUT value;  object;  (no default);  value of the attribute
REPEAT 2
OUTPUT output;  object;  the object with attributes set
```

The Options module is assigned to the Structuring category. It has three named parameters, none of which is given defaults. The module may take additional pairs of input parameters, whose types are the same as the last two inputs preceding the **REPEAT** statement.

10.2 Asynchronous Modules

Inboard, outboard, and runtime-loadable modules can be asynchronous. That is, depending on events external to Data Explorer, an asynchronous module can request that it be rerun. If Data Explorer is in execute-on-change mode, the module will reexecute immediately. If it is not, the module is called the next time the user runs the network.

To cause executions in this fashion, the ASYNC flag must be set in the module's **.mdf** file. Then it can call the DXReadyToRun() function to request reexecution. (For an example of how to use this function, see the sample outboard files "async.c" and "watchfile.c" in **/usr/lpp/dx/samples/outboard**.) If the module is outboard, the PERSISTENT flag must also be set, so that it does not exit after each execution.

DXReadyToRun() can be called in a variety of ways: by a signal handler (signal), after a prescribed time interval has passed (sleep, alarm), when a file appears (stat), or when data is received across a pipe or socket from another process (select).

If a module must wait for input associated with a file descriptor (e.g., a socket), it should use DXRegisterInputHandler to add another file descriptor to the select list. If data is received along with the input associated with that file descriptor, a user-supplied routine is called to check the status and may call DXReadyToRun. When the module is called, it can read the information in, process it, and return the output(s).

10.3 Inboard, Outboard, and Runtime-loadable Modules

The chief differences between inboard, outboard, and runtime-loadable modules lies in the following features:

- the module description file (**.mdf**)
- the compilation and linking process (i.e., the Makefile)
- the command that starts Data Explorer using the module.

Module description files are discussed in 10.1, "Module Description Files" on page 80. The other two features are discussed in the three sections following this brief summary.

inboard modules
Are compiled into Data Explorer. That is, the version of dxexec found (usually) in /usr/lpp/dx/bin_architecture is replaced with your own copy (i.e., a copy incorporating your module).

outboard modules
Run as separate processes. Linking an outboard module is quick, since it does not involve creating an entire new version of dxexec (as the compilation of inboard modules does). Thus an outboard module is also easier to debug because it can be relinked more quickly.

However, outboard modules are typically less efficient than other modules, especially if significant amounts of data must be transferred: data objects are transferred to and from an outboard module via sockets rather than as the pointers to shared memory that inboard and runtime-loadable modules use.

runtime-loadable modules
Can be loaded when Data Explorer is started or at any time after, and they do not require a separate copy of dxexec, as inboard modules do. Thus these modules have the advantage of portability without the disadvantage of the data-transfer overhead associated with outboard modules. A single executable can contain multiple modules that can be used like a library.

10.4 Compiling, Linking, and Debugging an Inboard Module

The following sample makefile templates for creating inboard modules can be found in **/usr/lpp/dx/samples/user**:

- RISC System/6000* Systems: **Makefile_inboard_ibm6000**
- Silicon Graphics**: **Makefile_inboard_sgi**
- Sun Microsystems**: **Makefile_inboard_solaris** or **Makefile_inboard_sun4**

- Hewlett-Packard**: **Makefile_inboard_hp700**
- Data General AViiON**: **Makefile_inboard_aviion**
- DEC Alpha**: **Makefile_inboard_alphax**

Replace **makex.o**, **add.o**, and **hello.o** with the names of your **.o** files. These makefiles assume that **user_inboard.mdf** is the name of the module description file that describes all your modules.

Starting Data Explorer requires specifying the module description file and a dxexec to the user interface:

```
dx -mdf my.mdf -exec mydxexec
```

Notes:

1. You can also load a .mdf file after Data Explorer has started. Use the **Load Module Description(s)** option in the **File** pull-down menu of the VPE window.
2. You must then restart the executive using the **Disconnect from Server** and **Start Server** options in the **Connection** pull-down menu of the VPE window (you would need to specify the dxexec, using **Options** in the **Start Server...** dialog box).

To debug a module you must first modify the CFLAGS line of the makefile to compile your source code as debuggable (**-g**) rather than optimized (**-O**).

Note: Data Explorer library routines are available only as optimized object code.

To debug a module:

1. Start up just the user interface: `dx -uionly`
2. Start a debugging session on your executable program.
3. Run the executable from the debugger with the **-r** (remote) flag.
4. Connect the user interface to the debugging session by selecting **Connect to already running server** in the **Options** dialog box of the **Start Server...** dialog box. You should check the port number specified when you start your executable from the debugging session, and ensure that the port number listed in the **Options** dialog box is the same.

10.5 Compiling, Linking, and Debugging an Outboard Module

The following sample makefile templates for creating outboard modules can be found in **/usr/lpp/dx/samples/user**:

- RISC System/6000* Systems: **Makefile_outboard_ibm6000**
- Silicon Graphics**: **Makefile_outboard_sgi**
- Sun Microsystems**: **Makefile_outboard_solaris** or **Makefile_outboard_sun4**
- Hewlett-Packard**: **Makefile_outboard_hp700**
- Data General AViiON**: **Makefile_outboard_aviion**
- DEC Alpha**: **Makefile_outboard_alphax**

Replace **makex.o**, **add.o**, and **hello.o** with the names of your **.o** files; replace **m_Hello**, etc. with the names of your modules; and replace **hello**, etc. with the names you want for your executables. The .mdf file for the outboard modules is **user_outboard.mdf**.

Linking Outboard Modules

Typically outboard modules are linked to the library dxlite, which contains the Data Explorer data model routines (see Appendix B, “Data Explorer Data Model Library: DXlite Routines” on page 181). This library does not contain all of the Data Explorer routines (see Appendix C, “Data Explorer Library Routines” on page 183), and an outboard module requiring access to such “additional” routines must be linked to the library dxcallm. However, the resulting outboard executable will be significantly larger than it would be otherwise.

Starting Data Explorer requires specifying the .mdf file to the user interface:

```
dx -mdf my.mdf
```

Notes:

1. You can also load a .mdf. file after Data Explorer has started. Use the **Load Module Description(s)** option in the **File** pull-down menu of the VPE window.
2. In script mode, Data Explorer does not recognize the -mdf flag, so you must add the following commands to your script before calling the module:

   ```
   Executive("mdf file", "module_name.mdf");
   $sync
   ```

To debug a module you must first modify the CFLAGS line of the makefile to compile your source code as debuggable (**-g**) rather than optimized (**-O**).

Note: Data Explorer library routines are available only as optimized object code.

To debug a module, start Data Explorer with the additional flag `-outboarddebug`. Instead of automatically starting the module, Data Explorer will prompt you to start the executable. You can then run the module from a debugger, using the flags specified to you by Data Explorer when it prompts you to start the module.

Special Considerations for Outboard Modules

Simple outboard modules

The simplest type of outboard module does not need to save information, does not communicate with any other process, and does not cause asynchronous executions. It takes inputs, computes something based on them, and returns outputs. The executable program that makes up the outboard module is run each time the module is called, and it exits after returning the output values.

Persistent outboard modules

To prevent the executable program of the outboard module from exiting after each execution, set the PERSISTENT flag on the FLAGS line in its **.mdf** file. This setting may be necessary so that the module can save information from one execution to another, or because repeated exits and restarts take too much time.

A persistent outboard module is started from the user interface the first time the module is called and does not exit until its icon is deleted from the network (program), the entire network is deleted, or the **Reset Server** option is selected from the **Connection** pull-down menu.

In script mode, persistent outboard modules are loaded the first time they are called and they do not exit until the executive exits or the command

```
Executive ("flush dictionary");
```

is run.

Global variables can be safely used to save information between executions of an outboard module. If the same outboard module occurs in a network more than once, a different process is started for each occurrence.

Note that the module may not be called at every execution: If the inputs are changed from their original values and then back again, Data Explorer saves the previous results and uses them without recalling the module. The “cache none” option prevents previous results from being saved, but you also need to set “cache none” for all downstream modules that process the outputs. Otherwise, caching at the lower levels will still prevent the module from being called each time. The SIDE_EFFECT flag specifies that the module is to be called each time, the performance penalty being that the module continues to execute even if the inputs remain unchanged.

Modules that can cause executions:

An asynchronous module can request that it be rerun. See 10.2, “Asynchronous Modules” on page 84.

Running an outboard on another machine:

If an outboard module should be run on one particular machine (perhaps because it is compute intensive and needs to run on a fast machine, or because it needs to access a peripheral that is connected to only one machine), the OUTBOARD line can specify a host name as well as an executable name. The Data Explorer libraries will take care of establishing a connection between where the main Data Explorer executive is running and the outboard host machine. The DXMODULES environment variable or the -modules flag can be used to specify a search path for outboard module executables, or the OUTBOARD line can specify a fully qualified path name.

A valid .rhosts file must be present to allow Data Explorer to use the “rsh” command to start a process on another machine. (See the UNIX manual page for “rsh” or “remsh” for more information.)

Miscellaneous information

DXReadyToRun cannot be called from the time a module receives its inputs until after it returns its outputs. To trigger another execution immediately, it can do so after the call to DXCallOutboard() in the outboard.c file.

Outboard modules cannot be written in coroutine style. They cannot produce outputs without being called by Data Explorer and thereby receiving new inputs (which can be ignored), and they must return something - the main Data Explorer executive will wait for the module to return before continuing.

An asynchronous module cannot be run in distributed mode, but it can be executed on another machine by setting the host name on the OUTBOARD line.

Asynchronous Outboard Module: An Example

The function of this example module is to monitor the status of a given file. Whenever the file is modified, its data are reimported. For example, this program could be used to monitor the output of a simulation program. The data can be plotted as they are created.

This sample program is /usr/lpp/dx/samples/outboard/watchfile.c. The same directory also holds the associated .mdf file (watchfile.mdf) and an example (watchsocket.c) that listens for input over a socket. See /usr/lpp/dx/samples/Outboard/Readme for more information abut sample modules.

```
/*
 * sample asynchronous outboard module.
 *
 * uses a signal to ask to be woken after a certain delay.
 * if a given file has been changed, re-import the data.
 *
 * see watchfile.mdf, which must be loaded before this can be run.
 * also see Makefile_architecture.name for how to compile this.
 */

#include <dx/dx.h>
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <sys/stat.h>


static Pointer  id = NULL;
static time_t   lastchanged;
static int      seconds;
static char     filename[1024];

/*
 * this routine is called each time the alarm signal is
 * issued.
 */
void signalcatch()
{
    struct stat Buffer;
    time_t changed;
    /* stat the file to find out its last modification time */
    if (stat(filename, &Buffer) == 0)
    {
      /* the last time the file was changed */
      changed = Buffer.st_mtime;

      /* compare to the last time the file was checked */
      if (lastchanged != changed)
      {
         /* the file has changed. Rerun the main program. */
         DXReadyToRun(id);
      }
      /* else the file hasn't changed since last time we checked */
      else
      {
         /* go back to sleep for some seconds, but first reset the
          * alarm */
         signal(SIGALRM, signalcatch);
         alarm(seconds);
      }
   }
}

Error
m_WatchFile(Object *in, Object *out)
{
    struct stat Buffer;
    char *file;

    /* the first input is the filename to check */
    if (!in[0])
    {
       DXSetError(ERROR_MISSING_DATA,"missing filename");
       return ERROR;
    }

    if (!DXExtractString(in[0], &file))
    {
       DXSetError(ERROR_BAD_PARAMETER,"filename must be a string");
       return ERROR;
    }

    /* put the filename into a static global variable */
    strcpy(filename,file);

    /* the second input is the number of seconds to wait between checks */
    /* the default is 10 seconds */
    if (!in[1])
       seconds = 10;
    else
    {
       if (!DXExtractInteger(in[1], &seconds))
       {
         DXSetError(ERROR_BAD_PARAMETER,"seconds must be an integer");
         return ERROR;
       }
    }
    /* the first time through, get the module id for the DXReadyToRun call */
    if (!id) {
     id = DXGetModuleId();
     if (!id) {
         out[0] = NULL;
         return ERROR;
     }
    }


    /* get the last modification time of the file */
    if (stat(filename, &Buffer) != 0) {
       DXSetError(ERROR_BAD_PARAMETER,"file %s not found");
       return ERROR;
    }

    lastchanged = Buffer.st_mtime;

    /* import the data from the file */
    out[0] = DXImportDX(filename, NULL, NULL, NULL, NULL);


    /* set the alarm for the next wakeup */
    signal(SIGALRM, signalcatch);
    alarm(seconds);


    return OK;
}
```

Note: If this program were compiled and linked as an inboard module, the global variables would have to be stored in the cache and associated with the module ID. Otherwise, the global variables would be shared among all calls to the module.

10.6 Compiling, Linking, and Debugging a Runtime-loadable Module

The following sample makefile templates for creating runtime-loadable modules can be found in **/usr/lpp/dx/samples/user**:

- RISC System/6000* Systems: **Makefile_loadable_ibm6000**
- Silicon Graphics**: **Makefile_loadable_sgi**
- Sun Microsystems**: **Makefile_loadable_solaris**
- Hewlett-Packard**: **Makefile_loadable_hp700**
- DEC Alpha**: **Makefile_loadable_alphax**

The makefile target “loadablelib” is an example of how to make a “library” of runtime-loadable modules. Replace **makex.o**, **add.o**, and **hello.o** with the names of your modules. The makefile target **hello** is an example of how to make an executable containing a single module. See also the files **hello_loadable.mdf** and **user_loadable.mdf**. Starting Data Explorer requires specifying the module description file to the user interface:

```
dx -mdf my.mdf
```

Note: You can also load a .mdf file after Data Explorer has started. Use the **Load Module Description(s)** option in the **File** pull-down menu of the VPE window.

To debug a module you must first modify the CFLAGS line of the makefile to compile your source code as debuggable (**-g**) rather than optimized (**-O**).

Note: Data Explorer library routines are available only as optimized object code.

To debug a module:

1. Start up just the user interface:

   ```
   dx -uionly -mdf yourmdf.mdf
   ```

2. Start a debugging session with the **-r** (remote) flag on `/usr/lpp/dx/bin_`*`workstation`*`/dxexec`.
3. Connect the user interface to the debugging session by selecting **Connect to already running server** in the **Options** dialog box of the **Start Server...** dialog box. You should check the port number specified when you start your executable from the debugging session, and ensure that the port number listed in the **Options** dialog box is the same.
4. Set your breakpoints in the debugger and continue.

   Note: On some architectures it may be necessary to build a module as inboard in order to debug it.

10.7 Memory Leaks

A memory leak will occur if the memory allocated by a module is not freed before that module returns its output(s). (See “Allocating and Freeing Memory” on page 13 for a list of Objects that typically need freeing.) Typically, if there is a memory leak, Data Explorer runs for some time. Then, after allocating all available memory, it stops executing and generates an error message that it is out of memory. It can resume execution only after the server has been disconnected and restarted.

You can check a module for a memory leak by running it several times, together with the Usage module. If the memory managed by the executive is flushed after each execution, memory that is allocated by a module and not freed before returning will cause the Usage module to report an increase in memory. The following Data Explorer script checks for a leak:

```
macro showleak()
{
   output = YourModule(input, ...);
   Print(output);
}
  ⋮
(any modules necessary to produce input for your module)
  ⋮

showleak();
showleak();
Executive(“flush cache”);
Usage(“memory”, 0);
showleak();
Executive(“flush cache”);
Usage(“memory”, 0);
showleak();
Executive(“flush cache”);
Usage(“memory”, 0);
```

Each call to Usage prints out the amount of memory used in both the small and the large arena. Unless there is a memory leak in the module, these amounts will remain constant.

Note: It is important to run the executable in “readahead off” mode. The amounts reported by Usage will be distorted if the Data Explorer executive reads the script ahead of execution. Specify:

```
dx -exec your_directory/your_executable -readahead off -script
```

Chapter 11. Working with Data Model Objects

This chapter describes the programming interface for creating and manipulating the basic Objects of the Data Explorer data model (some Objects are discussed in later chapters). For a brief summary of the data model, see Chapter 1, "Overview" on page 1. For a detailed description, see Chapter 3, "Understanding the Data Model" on page 15 in *IBM Visualization Data Explorer User's Guide*.

Data Explorer is an object-oriented graphical application. Its objects are data structures in global memory, they are passed by reference, and their contents are private to the implementation. See Table 1.

Table 1. Data Explorer Objects. The first column shows the hierarchical relationship of the Objects to one another. For subclasses Field *through* Private*, see 11.1, "Field Class" on page 97 through 11.5, "Private Class" on page 106 in this chapter; for* Interpolator*, see Chapter 13, "Data Processing" on page 131; and for the remainder, see Chapter 15, "Rendering" on page 149.*

Type	**Description**	**Class**
`Object`	Object Class	CLASS_OBJECT
`Field`	Data sampled on a regular or irregular grid	CLASS_FIELD
`Group`	Collections of Objects	CLASS_GROUP
Series	Time (or other) series	CLASS_SERIES
Multigrid	Group of Fields to be treated as one Field	CLASS_MULTIGRID
Composite Field	Group of Fields to be treated as one Field	CLASS_COMPOSITEFIELD
`Array`	Dynamic Arrays of data such as points	CLASS_ARRAY
Regular Array	One-dimensional series of evenly spaced points	CLASS_REGULARARRAY
Path Array	One-dimensional series of connected line segments	CLASS_PATHARRAY
Product Array	Regular or semi-regular grid positions	CLASS_PRODUCTARRAY
Mesh Array	Regular or semi-regular grid connections	CLASS_MESHARRAY
Constant Array	Array with a constant value	CLASS_CONSTANTARRAY
`String`	Object containing a string	CLASS_STRING
`Private`	Object pointing to private user data	CLASS_PRIVATE
`Interpolator`	Used to query Fields for data values	CLASS_INTERPOLATOR
`Xform`	Transformation matrix applied to an Object	CLASS_FORM
`Screen`	Object aligned to the screen	CLASS_SCREEN
`Clipped`	One Object clipped by another	CLASS_CLIPPED
`Camera`	Viewpoint, viewport, resolution	CLASS_CAMERA
`Light`	Lights	CLASS_LIGHT

Note: Any Group other than the three types listed here is a generic Group.
Any Array other than the five types listed here is an irregular Array.

11.1 Field Class

Each Field has some number of named components. Each component has a value (usually an Array) and some number of attributes, whose values are often strings or numbers. However, in the data model both components and attributes can be any Object. The defined components and attributes are listed in Chapter 3, “Understanding the Data Model” on page 15 in *IBM Visualization Data Explorer User’s Guide*.

`Field DXNewField()`
Creates a new Field Object. See Note on Use on page 97, and page 303.

`Field DXSetComponentValue()`
Adds a component to a Field. See Note on Use on page 97, and page 348.

`Field DXSetComponentAttribute()`
Adds or removes a named attribute from a component of a Field. See page 347.

`Object DXGetComponentValue()`
Returns a specified component of a Field. See Note on Use See page 243.

Note on Use

The following code segment illustrates how to make a component of one Field also a component of another Field:

```
f = DXNewField();
if (!f)
   return ERROR;
c = DXGetComponentValue(oldfield, "positions");
if (!DXSetComponentValue(f, "positions", c))
   return ERROR;
```

`Object DXGetComponentAttribute()`
Returns a named attribute of a specified component of a Field. See page 242.

`Object DXGetEnumeratedComponentValue()`
`Object DXGetEnumeratedComponentAttribute()`
Return a component or component attribute by index. These routines can be used to retrieve the components or component attributes when their names are not known. See page 246 and page 245.

`Field DXDeleteComponent()`
Deletes a named component from a Field. See page 221.

Less Commonly Used Routines

`Error DXComponentReq()`
`Error DXComponentOpt();`
`Error DXComponentReqLoc();`
`Error DXComponentOptLoc();`
Access or type-check a component in a Field. See Note on Use on page 98. See also page 209.

Note on Use

An example of the expected use of **DXComponentReq()** follows:

```
a = DXGetComponentValue(f, "positions");
if (!DXComponentReq(a, &points, &npoints, 0, TYPE_FLOAT, 3))
    return NULL;

a = DXGetComponentValue(f, "colors");
if (!DXComponentOpt(a, &colors, NULL, npoints, TYPE_FLOAT, 3))
    return NULL;
if (colors) ...
```

The first two statements check and retrieve a required “positions” component, while the next two check and retrieve an optional “colors” component that must have the same number of elements. Since the second call uses **DXComponentOpt()**, the program must check **colors** for **NULL** to determine whether the colors were present.

11.2 Group Class

This section summarizes the routines used with Groups, including those that manipulate members, Series Groups, MultiGrid Groups, Composite Field Groups, and parts of a Group. For a detailed description of Groups, see Chapter 3, “Understanding the Data Model” on page 15 in *IBM Visualization Data Explorer User’s Guide*.

Generic Operations

This section describes routines used to manipulate the members of a Group.

Group DXNewGroup()
Creates a new generic Group Object. See page 304.

Group DXSetMember()
Adds a member to a Group. See page 355.

Object DXGetMember()
Gets a named member of a Group. See page 255.

Group DXGetMemberCount()
Retrieves the number of members in a group. See page 256.

Object DXGetEnumeratedMember()
Returns the members of a Group by index. See page 247.

Group DXSetEnumeratedMember()
Adds a member to a Group by index. See page 350.

Group DXSetGroupType()
Group DXSetGroupTypeV();
Associates a type with a Group. See page 353.

Group DXUnsetGroupType()
Unsets the type associated with a Group. See page 371.

Class DXGetGroupClass()
Returns the subclass of a Group Object. See Note on Use. See page 251.

Note on Use

The following is an example of how this routine is used:

```
switch (DXGetGroupClass(g)) {
case CLASS_COMPOSITEFIELD
     ...
     break;
case CLASS_SERIES
     ...
     break;
}
```

Series Groups

Series Groups are a subclass of Group. A Series represents a single Field sampled across some parameter (e.g., a simulation of a CMOS device across a temperature range). Members of a Series have a position. A copy of the position is found in the "series position" attribute.

Every member of the Series must have the same dimensionality, the same data type, and the same connections element type. Members are stored in and retrieved from a Series Group by index rather than by name. Members cannot be retrieved by Series value.

`DXSeries NewSeries()`
Creates a new Series Object. See page 312.

`Series DXSetSeriesMember()`
Adds an indexed member to a Series Object. See page 361.

Note: **`DXSetMember()`** and **`DXSetEnumeratedMember()`** can also be used: the position is assumed to be the same as the sequence number of the member.

`Object DXGetSeriesMember()`
Returns an indexed member from a Series Object. See page 267.

Note: **`DXGetMember()`** and **`DXGetEnumeratedMember()`** can also be used for this purpose, but they do not return the **`position`** value.

MultiGrid Groups

A MultiGrid is a Group of Fields that is treated as a single entity. It is useful, for example, for holding certain kinds of simulation data represented by disjoint grids. All the members of a MultiGrid Group must have the same type of data and the same type of connection. However, unlike members of a Composite Field, Multigrid members are *not* required to be disjoint and abutting. The invalid-positions and invalid-connections components can be used to define which points of a grid are valid in a region of grid overlap.

`MultiGrid DXNewMultiGrid()`
Creates a new MultiGrid Object. See page 306.

Composite Fields

A Composite Field is a Group of Fields treated as a single entity. Parallelism in Data Explorer is achieved by explicitly partitioning Fields into a Composite Field. Composite Fields are typically created with the **`DXPartition`** routine (see page 316).

The connections component of each member must be of the same type, and the members are expected to be disjoint and abutting (i.e., sharing positions, data, etc., at the boundary).

`CompositeField DXNewCompositeField()`
Creates a new Composite Field Object. See page 301.

Parts

The Parts routines are provided to allow easier manipulation of all Fields in a Group Object without having to explicitly traverse the Object.

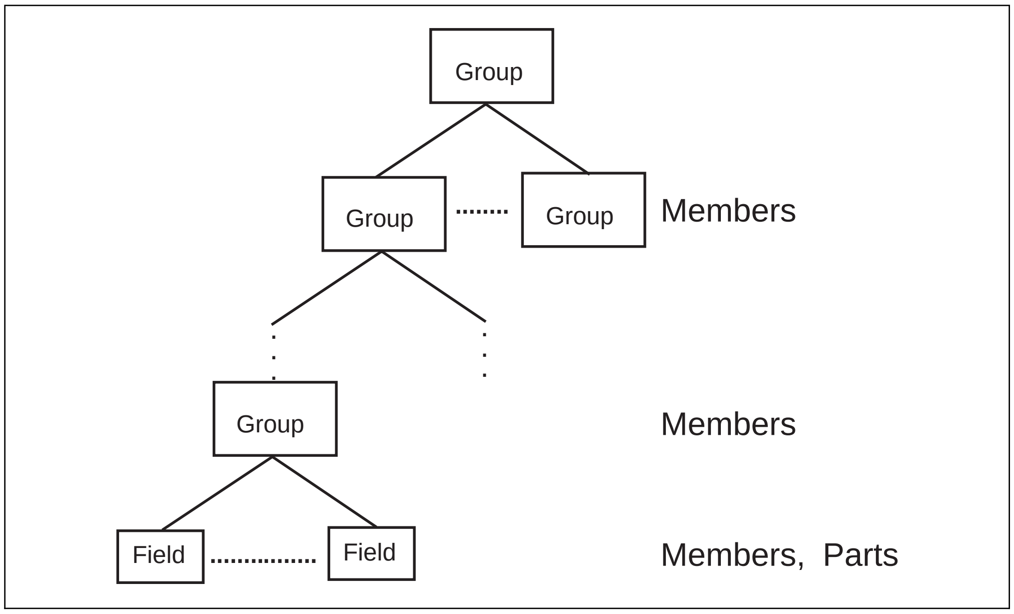


Figure 8. Parts of a Group

`Object DXProcessParts()`
Applies a function to every constituent Field (part) of a given Object. See page 320.

Note: **`DXGetPart()`**, **`DXGetPartClass()`**, and **`DXSetPart()`** are useful for prototyping and in cases where convenience outweighs efficiency. **`DXProcessParts()`** can often be used for the same purposes, and with greater efficiency.

`Field DXGetPart()`
Returns the parts of an Object by index. See page 261.

`Object DXGetPartClass()`
Returns by index only those sub-members of the given Group that are parts of a specified class. Note that **`DXGetPart(o, n)`** is equivalent to **`DXGetPartClass(o, n, CLASS_FIELD)`**. See page 262.

`Object DXSetPart()`
Sets a Field as a part of an Object. See page 357.

11.3 Array Class

Array Objects store user data, positions, connections, and photometric information (e.g., color or opacity).

Arrays may use explicit lists or one of several compact-coding schemes to store information. This section first describes the generic operations that are applicable to all Arrays, then operations specific to irregular Arrays, and finally operations specific to compact Arrays (i.e., regular, path, product, mesh, constant). For more information about Arrays, see Chapter 3, “Understanding the Data Model” on page 15 in *IBM Visualization Data Explorer User’s Guide*.

Generic Operations

Each Array **a** contains some number of items *n* (numbered *0* to *n-1*). Each item consists of a fixed number of elements of one type, specified when the array is created (see page 120 for the constants used to specify types).

Note: The routines listed in this subsection apply to both compact and irregular Arrays.

`Class DXGetArrayClass()`
Returns the subclass of an Array Object. See page 236.

`Array DXGetArrayInfo()`
Returns the number of items, type, category, rank, and shape of an Array. See page 239. (For information on rank and shape, see “Arrays” on page 28 in *IBM Visualization Data Explorer User’s Guide*.)

`Array DXTypeCheck()`
`Array DXTypeCheckV();`
Check that an Array matches a set of specifications. See page 369.

`Pointer DXGetArrayData()`
Returns a pointer to the start of a global memory area containing the items constituting the data stored in an Array. See page 237.

Note: To reduce memory requirements, it is preferable, where possible, to recognize compact arrays with **`DXGetArrayClass()`**, and not to expand them by calling **`DXGetArrayData()`**. An alternative is the set of Array-handling routines described in “Array Handling” on page 102.

`int DXGetItemSize()`
Returns the size in bytes of each individual item of an Array. See page 255.

`Pointer DXGetArrayDataLocal()`
Returns a pointer to the start of memory of a local copy of the data stored in an Array. See page 237.

`Array DXFreeArrayDataLocal()`
Frees space allocated by **`DXGetArrayDataLocal()`**. See page 233.

Irregular Arrays

Irregular Arrays are used for data that exhibit no particular regularity. They may also be used to manage dynamically growing collections of data whose size is not known in advance. **`DXNewArray()`** creates an irregular Array with no items; **`DXAddArrayData()`** adds data to an irregular Array; and **`DXGetArrayData()`** returns a pointer to an irregular Array.

Note: The routines listed in this subsection apply only to irregular arrays.

`Array DXNewArray()`
`Array DXNewArrayV();`
Create an irregular Array Object. See page 298.

`Array DXAddArrayData()`
Adds items to an Array. See Note on Use. See page 190.

`Array DXAllocateArray()`
Allocates space for the data items of an Array. Although this routine is not required, its use will make for more efficient management of memory. See page 197.

`Array DXTrim()`
Frees space previously allocated to an Array but not needed for the number of items in that Array. See page 368.

Note on Use

There are four ways to add data to irregular arrays.

1. Add the items one at a time: **`DXAddArrayData(a, i, 1, item);`**
2. Add the items in batches: **`DXAddArrayData(a, i, n, items);`**
3. Add the items all at once: **`DXAddArrayData(a, 0, n, items);`**
4. Allocate the memory as follows:
   a. call **`DXAddArrayData(a, 0, n, NULL)`**
   b. get a pointer to the memory:
   `ptr = DXGetArrayData(a)`
   c. put the items directly into global memory “by hand”: set the contents (pointed to by the pointer obtained in the preceding step) to the data value.

   ```
   ptr[i] = itemvalue;
   ```

   In the examples shown here:
   `a` is the array.
   `i` is the position at which to add an item.
   `n` is the number of items to be added.
   `item(s)` is the address of the item(s) to be added.

String List Routines

String lists are implemented as arrays of type **`TYPE_STRING, rank 1,`** and **`shape max_string_length+1`**. Each item should be a **`NULL`**-terminated character string.

`Array DXMakeStringList()`
`Array DXMakeStringListV();`
Create a String list from a given list of strings. See page 292.

Array Handling

Modules may have to handle a variety of different types of Arrays, such as constant, compact (e.g., regular or product), and irregular. **`DXGetArrayData()`** can be used on any of these types. However, if the original Array was compact, memory use is increased, sometimes dramatically.

The Array-handling routines simplify the task of dealing with the different types of Arrays at a cost in efficiency. Because it operates on a case-by-case basis, incremental methods available to **DXGetArrayData()** cannot be used. In addition, each element must be recomputed for each reference to that element. Therefore, multiple references to the same element will pay a penalty in execution time. However, if the array is irregular or constant, this interface can be substituted for the standard **DXGetArrayData()** with little degradation of performance.

The basic approach is to use **DXCreateArrayHandle()** for a given array, and then to retrieve the values of elements in that array, using either **DXIterateArray()**, **DXGetArrayEntry()**, or **DXGetArrayEntries()**.

DXCreateArrayHandle()
Creates a "handle" to allow convenient access to the items in any Array class. See page 214.

Error DXFreeArrayHandle()
Frees the memory allocated for an Array handle. See page 234.

Pointer DXGetArrayEntry()
Returns a specified item from an Array. See page 238.

void DXGetArrayEntries()
Returns specified items from an Array. See page 238.

Pointer DXIterateArray()
Iterates through an Array. See page 285.

Creating Positions and Connections Grids

Compact Arrays allow compact encoding of positions and connections. Four subclasses of Arrays represent 1- and multidimensional regular positions and connections:

	positions	**connections**
One-dimensional	RegularArray	PathArray
n-dimensional	ProductArray	MeshArray

In addition, the subclass Constant Array allows compact encoding of a constant value of any type, category, rank, or shape.

For more information about compact Array Objects, see Chapter 3, "Understanding the Data Model" on page 15 in *IBM Visualization Data Explorer User's Guide*.

Note on Use

The generic routine **DXGetArrayData()** expands the data of a compact Array into an explicitly indexed array and is the preferred means for this purpose. However, it is better still to code your algorithm so that no expansion of the Array is performed.

In addition to the low-level routines for creating various compact Arrays (described later), Data Explorer provides the following higher-level routines for creating a regular grid of positions or connections. These routines are to be preferred when there is a choice, because most Data Explorer functions support regular grids of positions or connections efficiently.

`Array DXMakeGridPositions()`
`Array DXMakeGridPositionsV();`
Create an *n*-dimensional grid of regularly spaced positions. See page 290.

`Array DXQueryGridPositions()`
Returns information about a regular positions grid. See page 327.

`Array DXMakeGridConnections()`
`Array DXMakeGridConnectionsV();`
Construct a grid of regular connections. See page 289.

`Array DXQueryGridConnections()`
Returns information about a regular connections grid. See page 327.

Regular Arrays

Regular Arrays encode linear regularity. A regular array is a set of *n* points lying on a line with a constant spacing between them, representing 1-dimensional regular positions. (The points themselves may be in a higher-dimensional space, but they must lie on a line.) All regular Arrays must be category real (as opposed to complex) and rank 1; the shape is then the dimensionality of the space in which the points are imbedded. (For information on rank and shape, see Chapter 3, “Understanding the Data Model” on page 15 in *IBM Visualization Data Explorer User’s Guide*.

`RegularArray DXNewRegularArray()`
Creates an Array containing evenly spaced data. See page 309.

`RegularArray DXGetRegularArrayInfo()`
Returns the number of items, the origin, and the delta of a regular Array. See page 266.

Path Arrays

Path arrays encode linear regularity of connections. A path array is a set of *n-1* line segments, where the *i*th line segment joins points *i* and *i+1*. All path arrays have type integer, category real, rank 1, and shape 2.

`PathArray DXNewPathArray()`
Creates an Array describing the connections between a set of points. See page 307.

`PathArray DXGetPathArrayInfo()`
Returns the number of points referred to in a path Array. See page 262.

Less Commonly Used Routines

`PathArray DXSetPathOffset()`
`PathArray DXGetPathOffset();`
Set and retrieve the offset value for the direction of the grid represented by this path. In the case where a path Array is used to define a regular grid of connections that is a part of a partitioned Field, it is useful to know the offset of the partition within the original Field. See page 357 and page 263.

Product Arrays

A product Array encodes higher-dimension Arrays as a Cartesian product of lower-dimension Arrays. The resulting set of positions constitutes an *n*-dimensional "grid" (i.e., a Cartesian product) derived from combining *n* Arrays.

Since each term is either regular or explicitly indexed, the resulting multidimensional positions are either completely or partially regular.

`ProductArray DXNewProductArray()`
`ProductArray DXNewProductArrayV();`
Create an Array that is the Cartesian product of a set of regular or irregular position Arrays. See page 308.

`ProductArray DXGetProductArrayInfo()`
Returns the number of terms and the terms of a product Array. See page 265.

Mesh Arrays

Mesh Arrays encode multidimensional regularity of connections. A mesh Array is a product of a set of connections Arrays. The product is a set of interpolation elements where the product has one interpolation element for each pair of interpolation elements in the two multiplicands, and the number of sample points in each interpolation element is the product of the number of sample points in each of the multiplicands' interpolation elements. This represents multidimensional regular connections. Each term may be either regular or not, resulting in either completely regular (for example, cubes) or partially regular (for example, prisms) multidimensional connections.

`MeshArray DXNewMeshArray()`
`MeshArray DXNewMeshArrayV();`
Create an Array that is the product of a set of regular or irregular connection Arrays. See page 306.

`MeshArray DXGetMeshArrayInfo()`
Returns the number of terms and the terms of a mesh Array. See page 256.

Less Commonly Used Routines

`MeshArray DXSetMeshOffsets()`
`MeshArray DXGetMeshOffsets();`
Set and retrieve the offset values along each dimension of a mesh. When a Mesh Array is used to define a regular grid of connections that is a part of a partitioned Field, it is useful to know the offset of the partition within the original Field. See page 355 and page 257.

Constant Arrays

Constant Arrays define Arrays that contain a number of items with the same value. These items may be of any type, category, rank, and shape.

`ConstantArray DXNewConstantArray()`
`ConstantArray DXNewConstantArrayV();`
Create an Array containing constant data. See page 302.

`Array DXQueryConstantArray()`
Determines if an Array contains constant data and, if so, returns number of items and data value. See page 325.

`Pointer DXGetConstantArrayData()`
Returns a pointer to the value stored in a Constant Array. See page 244.

11.4 String Class

String Objects encapsulate a **`NULL`**-terminated character string as an Object. For example, they can be used to associate a string-valued attribute with an Object. Since the value of an attribute must be an Object, a string-valued attribute is stored as a String Object. See “String List Routines” on page 102.

`String DXNewString()`
Creates a new String Object and initializes it with a copy of the specified **`NULL`**-terminated string. See page 312.

`char *DXGetString()`
Gets a pointer to the contents of a String Object. See page 268.

11.5 Private Class

Private Objects are an extension mechanism for storing data in the form of Objects. They are useful, for example, in the cache interface, where cached items must be Objects. The user is responsible for maintaining the data. In particular, the code creating a Private Object should also specify a deletion routine that will be called by the executive when the Object is deleted (to free, for example, any allocated private memory).

`Private DXNewPrivate()`
Creates an Object that contains private data. See page 308.

`Pointer DXGetPrivateData()`
Returns the private data pointer associated with a Private Object. See page 265.

11.6 Printing Objects

These routines show the hierarchy of the Object.

`Error DXPrint()`
`Error DXPrintV();`
Print an Object according to specified formatting options. See page 316.

11.7 Field Construction

This section describes routines that aid in the construction of fields.

Points and Dependent Data

Some Field components are often in one-to-one correspondence with the “positions” component (e.g., “data,” “colors,” “opacities,” and surface “normals”). (Alternatively, any of these components may be in one-to-one correspondence with the “connections” component.) If they are in one-to-one correspondence with positions (indicated by a “dep” attribute of “positions”), then they are also expected to be the same size as the positions component. The following routines aid in constructing such components.

Field DXAddPoint()
Field DXAddColor();
Field DXAddFrontColor();
Field DXAddBackColor();
Field DXAddOpacity();
Field DXAddNormal();

Field DXAddPoints();
Field DXAddColors();
Field DXAddFrontColors();
Field DXAddBackColors();
Field DXAddOpacities();
Field DXAddNormals();
Add points or point-dependent data to a Field. See page 194.

Field DXAddFaceNormal()
Field DXAddFaceNormals();
Add connection-dependent normals to a Field. See page 191.

Note on Use

The fourteen routines listed above are all suitable for adding a small number of points or for rapid prototyping; but for better performance, see **DXAddArrayData()** in “Irregular Arrays” on page 101.

Connections

The routines listed here define the interpolation elements of a Field, creating a “connections” component with an appropriate “element type” attribute.

Field DXAddLine()
Field DXAddTriangle();
Field DXAddQuad();
Field DXAddTetrahedron();
Add a single interpolation element to a Field. See page 192.

Field DXAddLines()
Field DXAddTriangles();
Field DXAddQuads();
Field DXAddTetrahedra();
Add interpolation element(s) to a Field. See page 192.

Field DXSetConnections()
Sets the “connections” component of a Field as a specified Array with a specified element type. See page 349.

Array DXGetConnections()
Gets the “connections” component of a Field and checks to see if it has a specified “element type” attribute. See page 244.

Standard Components

The following routines create and manipulate standard components of a Field.

Field DXEndField()
Creates the standard components that a Field is expected to contain if they do not already exist. See page 224.

Error DXEndObject()
Creates the standard components that Fields are expected to contain, sharing the results when components are shared between Fields. See page 226.

int DXEmptyField()
Determines whether a Field contains information. See page 224.

Field DXChangedComponentValues()
Field DXChangedComponentStructure();
Both routines delete all components of a Field that are derived from a specified component. **DXChangedComponentStructure()** also deletes all Field components that are dependent on or refer to a specified component. See Note on Use. See page 204.

Note on Use

This example illustrates one use of **DXChangedComponentValues()**:

```
g = DXCopy(f, COPY_STRUCTURE);
a = DXGetComponentValue(g, "positions");
b = ... modification of a ...
DXSetComponentValue(g, "positions", b);
DXChangedComponentValues(g, "positions");
```

Object DXBoundingBox()
Computes the bounding box of an Object. See page 201.

Array DXNeighbors()
Returns the neighbors Array of a Field. See page 297.

Error DXStatistics()
Returns statistical information for an Object. See page 365.

Object DXValidPositionsBoundingBox()
Computes the bounding box of the valid positions of an Object. See page 373.

11.8 Extracting Module Parameters

This section describes routines that aid in the parsing of parameters to modules. Inputs to modules that are simple items such as integers, floats, and character strings are packaged as Array Objects. The following routines simplify the extraction of such values. If the Object does not match (even if promoted as described in the following material), the routines return **NULL** but do not set the error code. Otherwise they return the original Object and fill in the pointer to the item.

If a float is expected, a byte, short, int, or long can be promoted to float. If an integer is expected, a byte or short can be promoted. If a float vector is expected, a byte, short, or integer vector can be promoted. If a string is expected, either a String Object or an Array of characters is accepted.

Object DXExtractInteger()
Determines whether an Object can be converted to an integer and, if so, extracts it. See Note on Use. See page 230.

Object DXExtractFloat()
Determines whether an Object can be converted to a floating-point value and, if so, extracts it. See page 229.

Object DXExtractString()
Determines whether an Object can be converted to a string and, if so, extracts it. See Note on Use. See page 232.

Object DXExtractNthString()
Determines whether an Object can be converted to a list of strings and, if so, extracts the *n*th one from it. See page 230.

Object DXQueryParameter()
Determines whether an Object can be converted to a specific value type. See page 330.

Object DXExtractParameter()
Determines whether an Object can be converted to a specific value type and, if so, returns the value in the user-provided buffer. See page 231.

Error DXQueryArrayConvert()
Error DXQueryArrayConvertV();
Determine if the given Array can be converted to an Array with the given type, category, rank, and shape. See page 325.

Error DXQueryArrayCommon()
Error DXQueryArrayCommonV();
Return a type, category, rank, and shape to which all of the arrays can be converted. See page 323.

Error DXArrayConvert()
Error DXArrayConvertV();
Create a new Array with a given type, category, rank, and shape from the data in the given Array. See page 199.

Array DXScalarConvert()
Converts the contents of an Array into scalar floating-point values. See page 342.

Note on Use

If a routine expects either a character string or an integer, the following code would determine the case and return the value.

```
Object o = input_object_to_check;
char *cp;
int i;

if (DXExtractInteger(o, &i))
    x = i;
else if (DXExtractString(o, &cp))
    strcpy(buffer, cp);
else
    DXSetError(...);
```

11.9 Creating Simple Data Explorer Objects

The following routines can be used to create a particular type of Data Explorer Object: integer, float, or string.

Array DXMakeFloat()
Returns a floating-point array containing a single floating-point value. See page 289.

Array DXMakeInteger()
Returns an integer array containing a single integer. See page 291.

String DXMakeString()
Returns a string object. See page 292.

11.10 Component Manipulation

This section describes advanced routines for explicitly manipulating Field components. Since these routines expect the input to be a copy of the Object, they operate directly on the input without making another copy. For additional information on components, see Chapter 3, “Understanding the Data Model” on page 15 in *IBM Visualization Data Explorer User’s Guide*.

Object DXRename()
Renames a component in a Field. See page 337.

Object DXSwap()
Exchanges two components in a Field. See page 365.

Object DXExtract()
Extracts a component from a Field. See page 228.

Object DXInsert()
Adds a component to a Field. See page 278.

Object DXReplace()
Adds a component from one Field to another. See page 338.

Object DXRemove()
Deletes components from a Field. See page 336.

Object DXExists()
Determines whether if a component exists in a Field. See page 227.

11.11 Data Import and Export

This section describes routines for importing and exporting data into and out of Data Explorer. This includes support for Data Explorer (`.dx`) files, as well as industry-standard netCDF files.

Data Explorer Format Files

Files in the Data Explorer format (see Chapter 3, “Understanding the Data Model” on page 15 in *IBM Visualization Data Explorer User’s Guide*) can be imported by the **DXImportDX()** routine and exported by the **DXExportDX()** routine.

Object DXImportDX()
Imports data from a Data Explorer file. See page 275.

Object DXExportDX()
Writes an Object to a specified file in a specified Data Explorer format. See page 228.

netCDF Data

Data Explorer can accept industry-standard netCDF files, which describe only a subset of the Data Explorer data model capabilities. For example, netCDF files are limited to regular data. This routine is provided for compatibility with other systems that use netCDF.

`Object DXImportNetCDF()`
  Imports data from a netCDF file. See page 276.

Chapter 12. System Services

This chapter describes the programming interface to basic system services: error handling and messages, timing, memory allocation, basic object services, types, Private Objects, String Objects, the cache, parallel programming, some basic convenience types and operations, module access, and asynchronous services.

For detailed descriptions of library routines see Appendix C, "Data Explorer Library Routines" on page 183.

12.1 Error Handling and Messages

Most Data Explorer library routines return either a pointer or an integer error code. A non-**NULL** pointer or the nonzero integer constant **OK** indicates success. **NULL** or **ERROR** (defined as zero) indicates failure.

If a library routine fails, it may use **DXSetError()** to set an error code. If it does, the (user-written) calling routine should return **NULL** or **ERROR** to propagate the error back to the user.

However, if the library routine does not set an error code, the calling routine should determine whether the **NULL** return indicates an error:

- If an error is indicated, the calling routine should set an error code (by calling **DXSetError()**) and return **NULL** or **ERROR**.
- If no error is indicated, the calling routine should proceed.

For example, **DXGetComponentValue()** returns **NULL** if the specified component does not exist, but it does not set an error code: the calling routine must determine whether the absence of that component is an error.

How any one Data Explorer routine handles error codes is described in the relevant entry in Appendix C, "Data Explorer Library Routines" on page 183.

The error codes are defined as follows:

```
typedef enum errorcode {
    ERROR_MIN,
    ERROR_NONE,
    ERROR_INTERNAL,
    ERROR_UNEXPECTED,
    ERROR_ASSERTION,
    ERROR_NOT_IMPLEMENTED,
    ERROR_NO_MEMORY,
    ERROR_BAD_CLASS,
    ERROR_BAD_TYPE,
    ERROR_NO_CAMERA,
    ERROR_MISSING_DATA,
    ERROR_INVALID_DATA,
    ERROR_BAD_PARAMETER,
    ERROR_MAX
} ErrorCode;
```

```
typedef int Error;
#define ERROR 0
#define OK 1
typedef void *Pointer;
#undef NULL
#define NULL 0
```

Error DXSetError()
#define DXErrorReturn()
#define DXErrorGoto()
#define DXASSERT()
Set an error code and an explanatory message. See Note on Use on page 115 and page 351.

Error DXAddMessage()
#define DXMessageReturn()
#define DXMessageGoto()
Append a message to the current error message. See Note on Use on page 115 and page 193.

void DXWarning()
Presents a warning message to the user. See page 374.

void DXMessage()
Presents an informational message to the user. See page 296.

Note on Use

When a function signals an error by returning **NULL**, it should also set an error code and an error message, using one of the following error routines:

1. **DXErrorReturn(errorcode, message);** Sets an error code and an error message, then returns **NULL**; this function should be invoked by the lowest-level routine that detects the error.
2. **DXMessageReturn(message);** Appends its own message to the message already recorded; this should be done by routines that:
   - Detect an error returned by a lower-level routine that has already set an error code.
   - Contain useful information to add to the message.
3. **return ERROR;** Is used when an error return is detected from a lower-level routine and the current routine has nothing useful to add to the message.

If cleanup is required before return, **DXErrorGoto()** or **DXMessageGoto()** may be used instead. Both routines require an **error:** label, after which cleanup is performed and either **NULL** (as shown here) or **ERROR** is returned.

```
error:
    ... cleanup goes here ...
    return NULL;
```

Less Commonly Used Routines

ErrorCode DXGetError()
Returns an error code for the last error that occurred. See page 247.

char *DXGetErrorMessage()
Returns the current error message. See page 249.

void DXResetError()
Resets the error state. See page 339.

void DXBeginLongMessage()
void DXEndLongMessage();
Together create a single “long” message from multiple calls to **DXMessage()**. See page 201.

void DXDebug()
void DXEnableDebug();
int DXQueryDebug();

Operate on global debug keys. See page 219.

DXDebug() compares an Array of keys to the global debug keys and calls **DXMessage()** if any are common to both.
DXEnableDebug() enables or disables one or more global debug keys.
DXQueryDebug() compares an Array of keys to the global debug keys and returns “1” if any are common to both.

12.2 Timing

The following routines involve “time marks,” used as the basis for measuring performance characteristics of the system.

DXMarkTime()
void DXMarkTimeLocal();

Record the times of various events in system operation (e.g., the beginning and the end of module execution). Time marks are batched until **DXPrintTimes()** is called. See page 295.

void DXPrintTimes()

Prints time marks. The accumulation and printing of timing messages must be enabled by **DXTraceTime()**. See page 318.

void DXTraceTime()

Enables or disables the accumulation of time marks. See page 367.

Note: For modules linked to Data Explorer, the Trace module enables the recording and printing of times (see Chapter 1, “Data Explorer Tools” on page 1 in *IBM Visualization Data Explorer User’s Reference*).

double DXGetTime()

Returns the elapsed time (in seconds) since system initialization. See page 269.

12.3 Memory Allocation

Data Explorer recognizes two kinds of memory—local and global. Stack variables and memory allocated by **DXAllocateLocal()** are local to the processor. Memory allocated by **DXAllocate()** (including all Objects) is global and may or may not reside in the processor. (On some platforms, such as those without per-processor local memory, there is no distinction between global and local.)

Data Explorer’s memory-allocation routines, which provide hooks for debugging, also distinguish between local and global allocation. These, rather than the standard system memory-allocation routines (such as **malloc()**), are recommended, to ensure consistent management of memory.

Pointer DXAllocate()
Pointer DXAllocateZero();

Allocate global memory. See page 196.

Pointer DXAllocateLocal()
Pointer DXAllocateLocalZero();

Allocate local memory if available; otherwise, global memory. See page 196.

Pointer DXAllocateLocalOnly()
Pointer DXAllocateLocalOnlyZero();
Allocate local memory. See page 196.

Pointer DXReAllocate()
Changes the size of a previously allocated block of memory. See page 334.

Error DXFree()
Frees a previously allocated block of memory. See page 233.

Less Commonly Used Routines

void DXPrintAlloc()
Prints out a summary of memory use. See page 317.

12.4 Object Class

The next four subsections summarize the type definitions and routines that apply to the Object class.

Type Definitions

An Object is represented by a pointer to a C structure stored in global memory. The content of the structure is private to the implementation. A **typedef** is provided for each class of Object:

```
typedef struct object *Object;
typedef struct string *String;
typedef struct private *Private;
typedef struct field *Field;

typedef struct group *Group;
typedef struct multigrid *Multigrid
typedef struct series *Series;
typedef struct compositefield *CompositeField;

typedef struct array *Array;
typedef struct regulararray *RegularArray;
typedef struct patharray *PathArray;
typedef struct productarray *ProductArray;
typedef struct mesharray *MeshArray;

typedef struct interpolator *Interpolator;

typedef struct xform *Xform;
typedef struct screen *Screen;
typedef struct clipped *Clipped;
typedef struct camera *Camera;
typedef struct light *Light;
```

An **enum** provides a number for each class and subclass:

```
typedef enum {
    CLASS_MIN,
    CLASS_OBJECT,
    CLASS_PRIVATE,
    CLASS_STRING,
    CLASS_FIELD,
    CLASS_GROUP,
    CLASS_MULTIGRID,
    CLASS_SERIES,
    CLASS_COMPOSITEFIELD,

    CLASS_ARRAY,
    CLASS_REGULARARRAY,
    CLASS_PATHARRAY,
    CLASS_PRODUCTARRAY,
    CLASS_MESHARRAY,

    CLASS_INTERPOLATOR,
    CLASS_FIELDINTERPOLATOR,
    CLASS_GROUPINTERPOLATOR,
    CLASS_LINESRR1DINTERPOLATOR,
    CLASS_LINESRI1DINTERPOLATOR,
    CLASS_QUADSRR2DINTERPOLATOR,
    CLASS_QUADSII2DINTERPOLATOR,
    CLASS_TRISRI2DINTERPOLATOR,
    CLASS_CUBESRRINTERPOLATOR,
    CLASS_CUBESIIINTERPOLATOR,
    CLASS_TETRASINTERPOLATOR,

    CLASS_GROUPITERATOR,
    CLASS_ITEMITERATOR,

    CLASS_XFORM,
    CLASS_SCREEN,
    CLASS_CLIPPED,
    CLASS_CAMERA,
    CLASS_LIGHT,
    CLASS_MAX,
    CLASS_DELETED
} Class;
```

Classes and Subclasses

Every Data Explorer Object is a member of **CLASS_OBJECT** and thus can be manipulated with generic-Object-class routines like **DXDelete()** and **DXGetObjectClass()**. Each Data Explorer Object is also a member of one of the subclasses of **CLASS_OBJECT** (e.g., **CLASS_FIELD**, **CLASS_GROUP**, and so on). Two of these subclasses (**CLASS_GROUP** and **CLASS_ARRAY**) themselves have subclasses (see Table 1 on page 96).

For any Object, **DXGetObjectClass()** returns the name of the “lowest” subclass to which that Object belongs. If an operation does not require a particular subclass of Object, it will not be affected if the Object is a not a member. If an operation does require a particular subclass, however, **DXGetGroupClass()** or **DXGetArrayClass()** can be used to identify it.

- For Groups, **DXGetGroupClass()** returns one of its three subclasses or (if the Group is generic) **CLASS_GROUP**.
- For Arrays, **DXGetArrayClass()** returns one of its five subclasses or (if the Array is generic) **CLASS_ARRAY**.

Any member of (superclass) **CLASS_ARRAY** or **CLASS_GROUP** can be manipulated with generic superclass routines such as **DXGetGroupType()** and **DXGetArrayInfo()**.

Object Routines

A number of routines can operate on any Object.

`Class DXGetObjectClass()`
: Returns the class of an Object. See page 260.

`Error DXDelete()`
: Deletes a reference to an Object. See page 220.

`Object DXSetAttribute()`
`Object DXDeleteAttribute();`
: Add or remove a named attribute from an Object. See page 344.

`Object DXGetAttribute()`
`Object DXGetEnumeratedAttribute();`
: Retrieve an attribute from an Object. See page 240 and page 245.

  The first retrieves a named attribute from an Object; the second, the *n*th attribute.

`Object DXGetFloatAttribute()`
: Retrieves a named attribute from an Object, verifies that its contents are a floating-point number, and returns that number. See page 249.

`Object DXGetIntegerAttribute()`
: Retrieves a named attribute from an Object, verifies that it contains an integer number, and returns that number. See page 253.

`Object DXGetStringAttribute()`
: Retrieves a named attribute from an Object, verifies that it contains a string, and returns a pointer to that string. See page 269.

`Object DXSetFloatAttribute()`
`Object DXSetIntegerAttribute();`
`Object DXSetStringAttribute();`
: Associate a floating-point, integer, or string attribute with an Object. See page 353, page 354, and page 362.

Objects can also be copied:

```
enum copy {
    COPY_ATTRIBUTES,
    COPY_HEADER,
    COPY_STRUCTURE
};
```

`Object DXCopy()`
: Copies the structure of an object. The **`COPY_`** parameter determines the depth to which an Object is copied:

  - **`COPY_ATTRIBUTES`** Creates a new Object of the same type as the old and copies all attributes and type information. It *does not* put references to members, components, etc. into the new object.
  - **`COPY_HEADER`** Copies only the header of the immediate Object. It *does not* copy the attributes, members, etc.; instead it puts references to them into the new Object.
  - **`COPY_STRUCTURE`** Is the parameter most frequently used with this routine.
    - For Groups: copies the Group header and recursively copies all Group members.

– For Fields: copies the Field header. It *does not* copy the components (which are usually Arrays); instead it puts references to the components into the resulting Field.
– For Arrays: passes back a pointer to the Array and makes no copy.

On error, the routine returns **NULL** and sets an error code. See "DXCopy" on page 211. See also Chapter 5, "Working with Positions" on page 37, Chapter 6, "Working with Connections" on page 43, and Chapter 7, "Importing Data" on page 47 for examples of this routine's use.

Less Commonly Used Routines

Object DXReference()
Increments the reference count of a specified Object. See page 335.

Error DXUnreference()
Removes a reference from an Object without deleting it. See page 371.

int DXGetObjectTag()
Object DXSetObjectTag();
Manipulate unique Object identifiers. See page 260.

Object DXCopyAttributes()
Copies attributes from one Object to another. See page 212.

Setting Data Types

The data type of Arrays, Fields, and Groups are determined as follows.

- **Arrays**: The data type is set when the Array is created. (See 11.3, "Array Class" on page 101.)
- **Fields**: The data type is that of the "data" component, if there is one.
- **Groups**: The data type is set explicitly by **DXSetGroupType()**; it is set implicitly for Series and Composite Groups (because members of these Groups must be of the same type).

```
typedef enum {
    TYPE_BYTE                (1 byte)
    TYPE_UBYTE               (2 bytes)
    TYPE_SHORT               (2 bytes)
    TYPE_USHORT              (2 bytes)
    TYPE_FLOAT               (4 bytes)
    TYPE_INT                 (4 bytes)
    TYPE_UINT                (4 bytes)
    TYPE_DOUBLE              (8 bytes)
    TYPE_HYPER               (8 bytes)
    TYPE_STRING              (dynamic)
} Type;

typedef enum {
    CATEGORY_REAL
    CATEGORY_COMPLEX
} Category;
```

Object DXGetType()
Returns the type, category, rank, and shape of an Object. See page 271.

int DXTypeSize()
int DXCategorySize();
The first returns the size (in bytes) of a variable of a given type; the second, the size multiplier for a given category. See page 370.

12.5 Cache

The look-aside cache service stores the results of computations for later use. Each cache entry is uniquely identified by a string function name, an integer key (which the executive uses to store multiple outputs for a single module), the number of input parameters, and the set of input parameter values.

The input parameters and the Object to be cached must be Data Explorer Objects.

Associating data with a cache entry. User data not already in the form of an Object can be associated with a cache entry by means of a Private Object (see 11.5, "Private Class" on page 106), encapsulating the data in an object. To associate more than one Object with a cache entry, use a Group to contain the Objects.

Losing a cache entry. Cache entries are subject to deletion without notice (e.g., when the system reclaims memory). The relative cost of creating an entry (which must be specified when the entry is created) may be taken into account when deleting Objects from the cache. If an estimate is not readily available, specify zero (0). Specifying **CACHE_PERMANENT** as the cost prevents the entry from being deleted during memory reclamation.

Object reference counts and the cache. It is important to be aware of the following: The system uses the cache to store intermediate results from modules. Thus inputs to modules often come from the cache. However:

- Once you have a pointer to a module's input, you can be sure it will not be deleted while you are processing it.
- If the input is a Group and you extract a member, that member will not be deleted while you are using it (because the Group will not be deleted).

However, Objects other than module inputs that are put into or retrieved from the cache behave differently: Once an Object is put in the cache, the system may delete it at any time to reclaim memory. This last has two consequences.

1. **DXGetCacheEntry()** returns an Object that is referenced so that it will not be deleted. Thus, you must delete the Object when you are finished using it:

   ```
   o = DXGetCacheEntry(...);
   ... use o ...
   DXDelete(o);
   ```

   Failure to do so will result in a memory leak, because the Object will always have an extra reference.
2. To continue using an Object after putting it in the cache, you must reference it *before* putting it there, and delete it when it is no longer needed:

   ```
   o = New...;
   DXReference(o);
   DXSetCacheEntry(..., o, ...);
   ... use o ...
   DXDelete(o);
   ```

Conversely, if putting the Object in the cache is the last operation before the return, and if **o** is not visible outside the scope of the routine, no reference is necessary:

```
o = New...;
... use o ...
DXSetCacheEntry(..., o, ...);
return ...;
```

where the return statement does *not* return **o**.

Cache Routines

#define CACHE_PERMANENT 1e32

Object DXGetCacheEntry()
Object DXGetCacheEntryV()
Retrieve a cache entry. See page 240.

Error DXSetCacheEntry()
Error DXSetCacheEntryV()
Set a cache entry. See page 345.

Error DXFreeModuleId()
Pointer DXGetModuleId()
Get a unique identifier for each instance of a module. See page 235 and page 258.

Error DXCompareModuleId()
Returns OK if the two specified module identifiers are the same; otherwise returns ERROR. See page 209.

Pointer DXCopyModuleId()
Returns a pointer to a copy of the specified module identifier. See page 213.

12.6 Pending Commands

In some cases, the module writer will want a particular task to be executed after the execution of a graph (visual program). This pending task can be specified and set for execution with DXSetPendingCmd.

DXSetPendingCmd()
Enters a task into a list of tasks to be run at the end of each graph execution. See page 358.

12.7 Looping Support

Two routines are supplied which affect the behavior of loops within Data Explorer. Loops are typically initiated with the ForEachMember or ForEachN modules. From within a user-written module, you could terminate a loop by calling **DXLoopDone**.

void DXLoopDone()
Terminates a loop. See page 287.

int DXLoopFirst()
Indicates whether it is the first time through a loop. See page 288.

12.8 Parallelism

Note on Use

Modules used exclusively in a uniprocessor environment do not require any of the routines described in this section. However, modules that use these routines on a parallel processor can also be run on a uniprocessor without changing any code.

Task Groups constitute a mechanism for specifying a collection of tasks to be performed in parallel on a multiprocessor. The task model provides simple fork/join semantics, suitable for coarse-grain parallelism:

1. Begin a collection of tasks to be executed in parallel (with **`DXCreateTaskGroup()`**)
2. Specify each task (with **`DXAddTask()`**)
3. Complete the task Group and begin execution (with **`DXExecuteTaskGroup()`**). Creating all the tasks first simplifies the model and allows optimal scheduling on the basis of estimated task-completion times.

Notes on Use

- It is important that all information required by parallel tasks be in global memory. This condition is generally met by passing Objects to tasks, since all Objects are in global memory.
- Tasks must not attempt to modify the same data structures simultaneously, with the exception of adding members to an existing Group (using **`DXSetGroupMember()`**).

`Error DXCreateTaskGroup()`
Starts a new Group of tasks to be run in parallel. See page 216.

`Error DXAddTask()`
Adds a task to be run later, in parallel if possible. See page 195.

`Error DXAbortTaskGroup()`
Aborts a task group without executing it. See page 189.

`Error DXExecuteTaskGroup()`
Runs the Group of tasks in the current Group in parallel, if possible. See page 226.

`int DXProcessors()`
Returns the number of processors. See page 320.

`int DXProcessorId()`
Returns the current processor identifier. See page 319.

12.9 Basic Data Types

This section describes some basic data types used by the system, including points, vectors, triangles, colors, and matrices.

Points and Vectors

Points are represented by the **Point** structure. Data Explorer provides a routine that constructs a point structure. Vectors are represented by the same structure as points, but for the sake of clarity they are defined as a separate type.

```
typedef struct point {
    float x, y, z;
} Point, Vector;

typedef int PointId;
```

Point DXPt()
Point DXVec();

Construct a Point or a Vector with the given coordinates. See page 322.

Lines, Triangles, Quadrilaterals, Tetrahedra, and Cubes

These data structures define the interpolation elements of an Object. They refer to points by point identifiers.

```
typedef struct line {
    PointId p, q;
} Line;

typedef struct triangle {
    PointId p, q, r;
} Triangle;

typedef struct quadrilateral {
    PointId p, q, r, s;
} Quadrilateral;

typedef struct tetrahedron {
    PointId p, q, r, s;
} Tetrahedron;

typedef struct cube {
    PointId p, q, r, s, t, u, v, w;
} Cube;
```

Figure 9 on page 125 shows the order of vertices in each structure. For more information about connections and the order of vertices, see Chapter 3, “Understanding the Data Model” on page 15 in *IBM Visualization Data Explorer User’s Guide*.

Line DXLn()
Triangle DXTri();
Quadrilateral DXQuad();
Tetrahedron DXTetra();

Construct a line, triangle, quadrilateral, and tetrahedron respectively, given the appropriate point identifiers. See page 286.

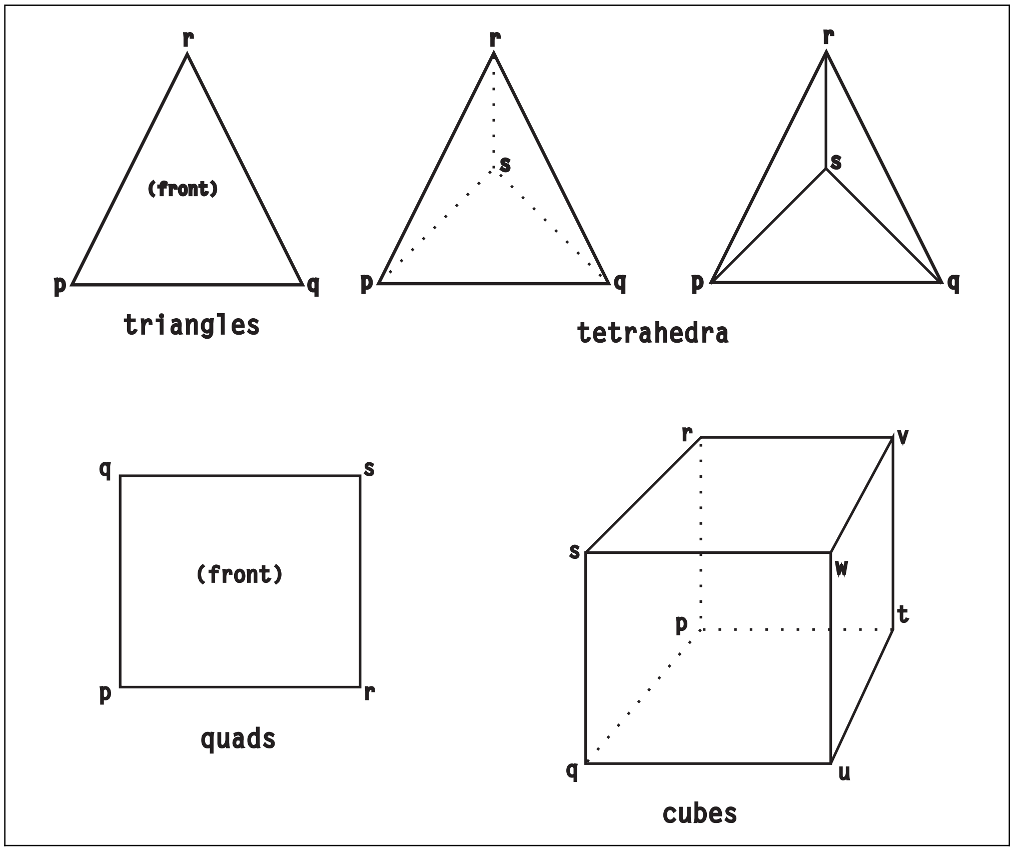


Figure 9. Order of Vertices in Connection Elements. In the tetrahedron at right, **s** *is the point nearest the viewer; in the tetrahedron at center, the point furthest from the viewer.*

Colors

Colors define the photometric characteristics of an Object; they may be associated with points, triangles, or lights, and may be used to define its reflectance or opacity. In Data Explorer, colors are floating-point numbers and therefore open ended, as is light intensity in the real world. However, real output devices can display only a limited range of intensities. In general, the range from 0.0 to 1.0 is by default mapped onto the range of displayable intensities of the output device, so colors should normally be specified in this range. Data Explorer provides a routine that constructs an RGB color structure.

```
typedef struct rgbcolor {
    float r, g, b;
} RGBColor;
```

RGBColor DXRGB()
Constructs an RGB color structure with the given components. See page 339.

Error DXColorNameToRGB()
Gets the RGB values for a specified colorname string. See page 208.

Angles

Angles are expressed as floating-point numbers in radians. The following macros express angles in units that might be more convenient. For example, **5*DEG** is 5 degrees in radians.

```
typedef double Angle;
#define DEG (6.283185307179586476925287/360)
#define RAD (1)
```

Transformation Matrices

Transformation matrices (or transforms) specify the relationship between Objects. For example, when a renderable object is included in a model, a transformation matrix specifies its placement. In Data Explorer, a transform is a 4×3 matrix specifying rotation and translation. This is a homogeneous matrix without the part that computes the *w* component of the result. (The *w* component is used for perspective, which is specified by a camera and is not needed here.)

```
typedef struct matrix {
    /* xA + b */
    float A[3][3];
    float b[3];
} Matrix;

static Matrix mdentity = {
    1, 0, 0,
    0, 1, 0,
    0, 0, 1,
    0, 0, 0
};
```

Transforms can be specified in a number of ways:

Matrix DXRotateX()
Matrix DXRotateY();
Matrix DXRotateZ();
Return a Matrix that specifies a rotation (in radians) about the *x*, *y* or *z* axis by angle **angle**. See page 340.

Matrix DXScale()
Returns a Matrix that specifies a scaling by amounts **x**, **y**, and **z** along the *x*, *y* and *z* axes. See page 340.

Matrix DXTranslate()
Returns a Matrix that specifies a translation by vector **v**. See page 340.

Matrix DXMat()
Returns a Matrix with the specified components. See page 340.

Basic Operations

A number of basic operations on the **Matrix**, **Point**, and **Vector** types are defined here. Operations declared as operating on type **Vector** also work on **Point** because both are type-defined for structure. These operations all take their arguments by value and return the result.

Vector DXNeg()
Vector DXNormalize();
double DXLength();
Perform unary operations of negation, normalization, and length. See page 189.

Vector DXAdd()
Vector DXSub();
Vector DXMin();

`Vector DXMax();`

Perform vector operations of addition, subtraction, min, and max. Min and max operations are performed on each component of a vector. See page 189.

`Vector DXMul();`
`Vector DXDiv();`

Multiply or divide a vector by a floating-point number. See page 189.

`float DXDot()`
`Vector DXCross();`

Form the dot product or cross-product of two vectors. See page 189.

`Matrix DXConcatenate()`

Returns a Matrix equivalent to the concatenation of two matrices. See page 210.

`Matrix DXInvert()`
`Matrix DXTranspose();`
`Matrix DXAdjointTranspose();`
`float DXDeterminant();`

Compute, respectively, the inverse, transpose, adjoint transpose, and determinant of a matrix. See page 210.

`Vector DXApply()`

Forms the product of a vector (interpreted as a row vector) and a matrix. See page 210.

12.10 Module Access

The module-access routines listed here enable the programmer to call other modules through an interface similar to the scripting language.

Modules are called by name, and parameters are specified as name-value pairs, freeing the programmer from having to supply values for all possible parameters. Optional parameters use the same defaults as they would if being executed directly by the executive. (If other parameters are added in subsequent releases, the call remains upwardly compatible.)

Note: The following modules cannot be called by DXCallModule:

- Interactors

FileSelector	Integer	IntegerList	Reset
Scalar	ScalarList	Selector	SelectorList
String	StringList	Toggle	Value
ValueList	Vector	VectorList	

- Flow Control

Done	Execute	First	ForEachMember
ForEachN	GetGlobal	GetLocal	Route
SetGlobal	SetLocal	Switch	

- Interface Control

ManageColormapEditor	ManageControlPanel	ManageImageWindow	ManageSequencer

- Special

Colormap	Sequencer	

- DXLink

DXLInput	DXLInputNamed	DXLOutput

Data Explorer modules can be called by inboard, outboard, and runtime-loadable modules linked to Data Explorer. They can also be called by stand-alone programs (for examples, see /usr/lpp/dx/samples/callmodule).

Note: Use of DXCallModule in a stand-alone program or outboard module requires linking to the library libDXcallm.a.

```
 Error DXCallModule();
object DXModSetFloatInput();
object DXModSetIntegerInput();
  void DXModSetObjectInput();
  void DXModSetObjectOutput();
object DXModSetStringInput();
       DXSetModuleInput();
       DXSetModuleOutput();
```

Allow a module to call another module (see Note on Use). See page 203.

`Void DXInitModules()`
Must be called when using DXCallModule in a stand-alone program or outboard module. See page 277.

`Void DXSetErrorExit()`
Determines the action taken when DXSetError is called by a stand-alone program. See page 352.

`Void DXGetErrorExit()`
Returns the current error-handling level as set by DXSetErrorExit. See page 248.

`Error DXCheckRIH()`
Checks registered input handlers. See page 205.

Note on Use

This example calls the Slab module.

```
Error Slab1(Object toBeSlabbed, int dimension, int position
      Object *slabbedObject)
{

    ModuleInput in[3];
    ModuleOutput out[1];
    Error result;

    DXModSetObjectInput(&in[0], "input", toBeSlabbed);
    DXModSetIntegerInput(&in[1], "dimension", dimension);
    DXModSetIntegerInput(&in[2], "position", position);

    DXModSetObjectOutput(&out[0], "output", SlabbedObject);

    result = DXCallModule("Slab", 3, in, 1, out);

    return result;
}
```

12.11 Asynchronous Services

The asynchronous services routines enable a module with an external interface (e.g., to another application through a socket) to signal the executive that it is ready to run again.

If the module has signaled its readiness to run again, and Data Explorer is in **Execute on Change** mode, the system reexecutes the module immediately (and any modules downstream as well). Otherwise, it does not run until the next time the user initiates an execution.

Error DXReadyToRun()
Enables an asynchronous module to signal that it is ready to execute again. See page 334.

Error DXRegisterInputHandler()
Assigns a handler routine for input coming from an open file descriptor. See page 336.

Error DXFreeModuleId()
Pointer DXGetModuleId()
Get a unique identifier for each instance of a module. See page 235 and page 258.

Error DXCompareModuleId()
Returns OK if the two specified module identifiers are the same; otherwise returns ERROR. See page 209.

Pointer DXCopyModuleId()
Returns a pointer to a copy of the specified module identifier. See page 213.

Chapter 13. Data Processing

The higher-level processing functions available in Data Explorer include: data partitioning, interpolation and mapping, invalid-data handling, growing and shrinking partitioned data, hashing, and picking.

13.1 Data Partitioning

Partitioning is the process of dividing a Field into spatially disjoint subsets called “partitions.” The result is a *Composite Field*, whose members are partitions. Partitioning is particularly useful for parallel processing.

Since each interpolation element of an input Field is assigned to one and only one partition, the resulting partitions cover precisely the same region of space as the original input Field did without overlap. However, because the input elements cover the same region of space, the bounding boxes of the resulting partitions may overlap.

`Group DXPartition()`
Divides a Field into partitions. See page 316.

13.2 Interpolation and Mapping

The interpolation service described here performs linear interpolations on the values of a Field or Composite Field. The values are those of a function $y(x)$ defined in terms of points x_i (“positions”), values y_i (“data”), and basis functions $b_i(x)$ (implicitly defined by the element type or by the faces).

Interpolator Objects are the interface to calling applications. They:

- Provide interpolation methods that are appropriate to the values in the Field Object operated on. The choice of method is based on factors such as the hierarchical structure of the data model, data formats, primitive types, and interpolation model.
- Contain data structures that facilitate interpolation. These structures are initialized either:
  - immediately, when the interpolator is created, or
  - on demand, during the interpolation process. This delayed initialization is especially useful when the data being interpolated is partitioned and only some of the partitions require interpolation. However, if the interpolator is intended for sharing, it must be fully initialized before it is copied. This initialization can be done in parallel prior to the creation of subtasks.
- Use information gathered in previous interpolations to speed subsequent interpolations. For that reason, Interpolator Objects contain data specific to the process that uses the interpolator. Consequently, each parallel process must be provided with its own interpolator. The most efficient way of creating such “individualized” interpolators is to generate a single, fully initialized interpolator; pass it to the parallel subtasks that need to use it; and have the subtasks then *copy* that interpolator for local operation. When this approach is used, the parent interpolator cannot delay initialization: it must be fully initialized before it is copied.

Note: Fields interpolated through this interface must have the same dimensionality as the space in which they are embedded. For example, triangles embedded in a 2-dimensional plane can be interpolated; triangles in 3-dimensional space cannot. For interpolation, faces must link 2-dimensional positions, and data must be dependent on “faces.”

```
enum interp_init {
    INTERP_INIT_DELAY,
    INTERP_INIT_IMMEDIATE,
    INTERP_INIT_PARALLEL
};
```

Interpolator DXNewInterpolator()
Creates an Interpolator Object. See page 305.

Interpolator DXInterpolate()
Interpolates data values in a Field. See page 280.

Interpolator DXLocalizeInterpolator()
Copies the interpolator structures into local memory. See page 287.

Object DXMap()
Interpolates data values at sample points. See page 293.

Array DXMapArray()
Provides an intermediate-level mapping function. See page 294.

Object DXMapCheck()
Verifies that Objects are valid for mapping. See page 294.

13.3 Invalid Data

In order to support the elimination of unwanted items from a data set, positions, connections, faces, or polylines in a Field can be “invalidated” by placing a corresponding “invalid positions,” “invalid connections,” “invalid faces,” or “invalid polylines” component in that Field. An “invalid” component lists the individual data items of the positions, connections, faces, or polylines component that are invalid.

In position-, connection-, face-, and polyline-dependent components, data items that correspond to invalid elements are themselves invalid. Data may be invalidated by modifying (or creating, if necessary) these invalid components. If no invalid component exists, all data items are assumed to be valid.

An invalid component is represented by one of two types of Array:

1. An Array of length equal to the length of the “positions,” “connections,” “faces,” or “polylines” component. This component has the “dep” attribute of “positions,” “connections,” “faces,” or “polylines” (i.e., the elements correspond one-to-one with the elements in the applicable Array). The Array is either **TYPE_BYTE** or **TYPE_UBYTE**; its valid elements have a value of **DATA_VALID** (0), invalid elements a value of **DATA_INVALID** (1).
2. An Array of length equal to the number of invalid members of the applicable component. This component has the “ref” attribute of “positions,” “connections,” “faces,” or “polylines.” The Array is either **TYPE_INT** or **TYPE_UINT**, and the indices of the invalid members are listed.

If many elements of the component are invalid, the first type of Array is preferable. If only a few elements are invalid, the second is preferable.

The routines described here simplify the handling of both types of invalid component.

Once positions have been invalidated, their immediate connections, faces, or polylines can also be invalidated by calling **`DXInvalidateConnections()`**, which will create an “invalid connections,” “invalid faces,” or “invalid polylines” component if necessary. (Note that this component cannot be assumed to be up-to-date unless this routine is called.)

Invalid positions, connections, faces, and polylines (and their dependent information) can be removed from the data set by calling **`DXCull()`**. This routine:

- Removes all invalidated elements and the corresponding elements of components that are dependent on invalidated elements.
- Renumbers components that reference positions, connections, faces, and polylines (inserting a -1 for indices that refer to removed positions, connections, faces, and polylines).
- Removes invalid positions, invalid connections, invalid faces, and invalid polylines components.

Removal of invalid components may affect system performance significantly if it requires the conversion of regular positions and connections components to irregular form (i.e., by greatly increasing the memory used for these components). For example, **`DXCreateInvalidComponentHandle()`** creates a “handle” for a specified invalid component. Other routines set particular elements as valid or invalid or they determine the validity of a given element. The module writer can set up a handle to mark elements as either valid or invalid (e.g., initializing all elements to invalid and validating the appropriate elements or vice versa).

When the time comes to create an invalid-component Array, the information stored in the handle is used to create one of the two kinds of Array just described, depending on the relative number of invalid elements.

Note: Before adding a new invalid component to a Field, it is important to explicitly remove any invalid component having the same name. The reason for this requirement is that the attributes of an existing component will be copied to the new component. Overwriting a “dep” invalid component with a “ref” invalid component will result in a component with *both* attributes, which is self-contradictory.

```
#define DATA_VALID   0
#define DATA_INVALID 1
```

`Object DXInvalidateConnections()`
Propagates the invalidity of positions. See page 280.

`Object DXInvalidateDupBoundary()`
Invalidates all but one of the positions shared between partitions in a Composite Field. See page 281.

`Object DXInvalidateUnreferencedPositions()`
Determines which positions in the Fields of the input Object are not referenced by any connections element and invalidates them. See page 281.

`Object DXCull()`
Removes invalid positions and connections (and their dependent information) from an Object. See page 217.

`InvalidComponentHandle DXCreateInvalidComponentHandle()`
Creates an invalid-component handle. See “Examples” on page 136. See page 215.

Error DXFreeInvalidComponentHandle()
: Frees all memory associated with an invalid-component handle. See page 234.

Error DXSaveInvalidComponent()
: Creates a new invalid-component Array containing the information stored in an invalid-component handle and stores it in a given Field. See "Examples" on page 136. See page 342.

Array DXGetInvalidComponentArray()
: Returns an Array containing the information stored in an invalid-component handle. See page 254.

Error DXSetElementValid()
: Sets the validity state of a specified element in an invalid-component handle to **DATA_VALID**. See page 350.

Error DXSetElementInvalid()
: Sets the validity state of a specified element in an invalid-component handle to **DATA_INVALID**. See "Examples" on page 136. See page 349.

int DXIsElementValid()
int DXIsElementInvalid();
: Return the validity state of a specified element of an invalid-component handle. See "Examples" on page 136. See page 283.

int DXIsElementValidSequential()
int DXIsElementInvalidSequential();
: Return the validity state of a specified element of an invalid-component handle when the queries come in sequential order. See page 283.

int DXGetValidCount()
: Returns the number of valid elements in an invalid-component handle. See page 271.

int DXGetInvalidCount()
: Returns the number of invalid elements in an invalid-component handle. See page 254.

Error DXSetAllValid()
: Sets all elements valid. See page 343.

Error DXSetAllInvalid()
: Sets all elements invalid. See "Examples" on page 136. See page 343.

Error DXInvertValidity()
: Reverses the validity state of every element in a specified invalid-component handle. See page 282.

Error DXInitGetNextInvalidElementIndex()
Error DXInitGetNextValidElementIndex();
: Prepare an invalid-component handle for iteration through the invalid or valid elements. See page 278.

int DXGetNextInvalidElementIndex()
: Returns the index of the next invalid element after the index returned on the previous call. See page 259.

int DXGetNextValidElementIndex()
: Returns the index of the next valid element after the index returned on the previous call. See page 259.

Examples

Invalid-component handles have a variety of uses, as shown in these examples.

- The call to DXCreate... creates an invalid-component handle that stores the validity state of the data associated with the positions. If there is an initial “invalid positions” component, its contents initialize the handle.

  ```
  handle = DXCreateInvalidComponentHandle(field, NULL, “positions”)
  ```

- Here the call to DXCreate... creates an invalid-component handle associated with **array** (in this case a connections component). No initialization takes place; the handle is initialized to “all valid.”

  ```
     array = (Array)DXGetComponentValue(field, NULL, “connections”);
     handle = DXCreateInvalidComponentHandle(array, NULL, NULL);
  ```

  Note that in this example DXCreate... has no way of determining the component name of **array** and, when the handle is converted to an Array, cannot attach a “dep” or “ref” attribute. It is therefore the caller’s responsibility to attach the appropriate attribute before placing the Array in a Field. You can determine whether a “dep” or “ref” attribute is needed by examining the type of the invalid Array: **TYPE_INT** or **TYPE_UINT** implies references. As noted earlier, if this component is added to a Field, any previous component of the same name must be explicitly deleted.

  Since it is easier to create an invalid-component handle from a Field and component name (as in the first example), **DXSaveInvalidComponent()** can be used to add the modified validity information to the Field.

- Here the call to DXCreate... creates an invalid-component handle associated with the positions component and initializes it with the invalid-positions Array.

  ```
     array = (Array)DXGetComponentValue(field, “positions”);
     iarray = (Array)DXGetComponentValue(field, “invalid positions”);
     handle = DXCreateInvalidComponentHandle(array, iarray, NULL);
  ```

  This example is similar in concept to the first except that it prevents the user from calling **DXSaveInvalidComponent()**.

- This example performs part of the Include operation: it invalidates all connections that reference an invalid position. Any invalid connections remain.

  ```
     inv_pos_h = DXCreateInvalidComponentHandle(field, NULL, “positions”);
     inv_con_h = DXCreateInvalidComponentHandle(field, NULL, “connections”);
     if (!inv_pos_h || !inv_con_h)
        goto error;

     for (i = 0; i < nConnections; i++)
     {
        elt = (int *) DXCalculateArrayEntry(array_handle, i, scratch);
        for (j=0; j < vertsPerElement; j++)
           if (DXIsElementInvalid(inv_pos_h, elt[j]))
           {
               DXSetElementInvalid(inv_con_h, i);
               break;
           }
     }
     if (!DXSaveInvalidComponent(field, inv_con_h))
        goto error:
  ```

- The following example initially invalidates all connections, and then, for each point, validates the connection that contains it. “IndexOfElementContainingPoint” is a user-supplied routine.

```
inv_con_h = DXCreateInvalidComponentHandle(field, NULL, “connections”);
if (!inv_con_h)
   goto error;

DXSetAllInvalid(inv_con_h);

for (i=0; i < nPoints; i++)
{
   j = IndexOfElementContainingPoint(field, point[i]);
   DXSetElementInvalid(inv_con_h, j);
         break;
}

if (!DXSaveInvalidComponent(field, inv_con_h))
   goto error:
```

13.4 Growing and Shrinking Partitioned Data

The routines listed at the end of this section are necessary for processing Composite Fields.

Some modules (e.g., filters) require information from the neighborhood of each point. Since partitioning divides data into spatially disjoint subsets for independent processing, a neighborhood may be divided among different partitions: for example, a filter kernel may overlap the boundary between two partitions. In such cases, processing one partition requires information that resides in the other.

In order to facilitate such information sharing, Data Explorer includes routines that support temporarily overlapping partitions. **DXGrow()** modifies its input Field and adds to each partition information from the partition’s neighbor(s).

Because **DXGrow()** modifies its input, the calling routine must use **DXCopy()** to copy the input structure if that structure is not to be modified. After this boundary information has been accrued, the processing of the partition may be handled independently since all information required to produce correct results for the original partition is available in it. For example, in the case of filtering, boundary information is added so that wherever a filter kernel is placed in the original partition, the kernel does not extend outside the grown partition, producing correct results in the original partition. After processing the Field produced by **DXGrow()**, **DXShrink()** must be called to shrink any components that have not been shrunk by the caller, and to remove extra references to the original components that were put in the Field by **DXGrow()**.

When **DXGrow()** is called, the depth of an overlap region is specified by specifying the number of *rings* to be accrued. An element is said to be in the *k*th ring if it has at least one vertex in the *k*th ring. A vertex is in the 0th ring if it exists both in the partition and the neighbor, and is in the *k*th ring if it is not in a lower ring and an element in ring *k-1* is incident upon it. Most frequently, such modules produce results for each vertex on the basis of the elements incident on that vertex; this is achieved by requesting that **DXGrow()** include 1 ring: those elements from neighboring partitions that are incident on vertices that exist in both partitions.

The treatment of the exterior boundary of regular grid data is specified by a parameter to **DXGrow()**. You may specify that the Field not be expanded beyond its boundary (i.e., that the exterior partitions not be expanded except on the sides that border other partitions). Alternatively you may specify that the Field be expanded beyond its original boundaries, with the new data being filled in one of three ways: with a constant value; with the replicated value from the nearest edge point in the original Field; or with nothing, only reserving space for the new data but leaving its contents undefined.

While it is necessary that the footprint of a filter kernel, placed anywhere in the original partition, not extend past the grown partition boundary, it is probably not necessary to apply the filter in the boundary regions accrued from neighbors; these regions are properly handled during the processing of the neighboring partition. Data Explorer also includes routines that query the original number of positions and connections (in the case of irregular grids) or the offset relative to the grown partition and size of the original partition.

Frequently, modules do not require all components of a Field that are dependent on the positions to be grown. To avoid accruing information that will not be required during processing, **DXGrow()** requires the calling application to specify which components, in addition to positions and connections, will be required.

Modules using **DXGrow()** have the option of producing results corresponding to the positions of the larger grown Field or, more efficiently, producing results corresponding only to positions of the original smaller Field. Even though the former method is less efficient, involving more data movement and perhaps more calculation, it is sometimes more convenient. Therefore, the **DXShrink()** function is provided to shrink all components that depend on or reference positions or connections back to their original size. If the user has already shrunk the positions, **DXShrink()** will leave them unmodified. In any case, the **DXShrink()** function must be called after operating on a grown Field in order to remove references to the "original" components that were placed in the Field by **DXGrow()** for later use by **DXShrink()**.

For each component specified in the component list passed to **DXGrow()**, a component named "original *componentname*" is created. **DXShrink()** will rename each of these to its original name. Therefore, for components you have modified (e.g., data), you should remove the corresponding original component ("original data" in this example) before calling **DXShrink()**.

Both **DXGrow()** and **DXShrink()** operate in parallel on Composite Fields. For that reason, **DXGrow()** must be called prior to any subtasking invoked explicitly by the calling application; **DXShrink()** must be called after any such subtasking has been completed.

```
#define GROW_NONE      NULL
#define GROW_REPLICATE ((Pointer)1)
#define GROW_NOFILL    ((Pointer)2)
```

Object DXGrow()
Object DXGrowV();

Add information from neighboring partitions to a Composite Field. See page 273.

`Field DXQueryOriginalSizes()`
`Field DXQueryOriginalMeshExtents();`
Return information about the size of the original Field used as the input to **`DXGrow()`**. See page 329.

`Object DXShrink()`
Removes information added to an Object by **`DXGrow()`**. See page 364.

13.5 Hashing

This section describes a set of routines for storing an arbitrary number of elements with a fixed access time. This implementation is designed for general-purpose use in many applications. Copies of the elements are stored in a hash table.

The elements may be of any fixed size, set at the time that the hash table is created. Each element contains a key identifying the element, along with whatever data you choose to associate with that key. For example, a key might be an x, y, z point, with an associated data value for that point.

Elements are stored in the table using long integer pseudokeys. These pseudokeys should be uniformly distributed in any range beginning at zero.

Note: Pseudokey 0xFFFFFFFF is reserved. Items cannot not be placed in the hash table using this pseudokey value.

The elements themselves may contain the pseudokey as their first long integer word. Alternatively, the pseudokey may be derived from the element through a call-back function provided at the time the hash table is created.

More than one element may be stored under the same pseudokey if a compare function is provided at the time the hash table was created. Whenever the hash table query function is called with the same search key, the hash table is searched for an element whose pseudokey matches the key either in or derived from the search key. If no compare function has been provided, the found element is returned. However, if a compare function has been provided, it is called by the hash table query routine to match the search key against each element in the hash table that matches the pseudokey. When the compare function succeeds (returns a 0), the element is returned.

A similar sequence is used to either insert a unique element (if a compare function was provided) or to overwrite a previously inserted element of the same key (if a compare function was not provided).

Note: Only 16 elements may be stored using the same pseudokey.

`HashTable DXCreateHash()`
Creates a hash table. See “Examples” on page 140. See page 214.

`Element DXQueryHashElement()`
Searches a hash table for an element matching a specified key. See “Examples” on page 140. See page 328.

`Error DXInsertHashElement()`
Inserts an element into a hash table. See “Examples” on page 140. See page 279.

Error DXDeleteHashElement()
Removes any element that matches a search key. See page 221.

Error DXInitGetNextHashElement()
Initializes the pointer to the next element for **GetNextHashElement**. See page 277.

Error DXGetNextHashElement()
Returns the next element in a hash table. See page 258.

Error DXDestroyHash()
Deletes a hash table. See page 222.

Optional routines provided by the caller at the time of creation of the hash table follow:

hashFunc()
Converts a search key to a long integer pseudokey. Called on insertion and query.

cmpFunc()
Determines whether elements with the same pseudokey are unique. Called on insertion and query.

Examples

In the following examples, underscored items are supplied by the user.

(1) No hash or compare function is provided at the time the hash table is created. Stored elements are x, y, z points, along with associated data values.

Note: Because no hash function is provided, the pseudokey must be stored as the first long integer word of the element.

```
typedef struct
{
   long pseudokey;
   Point pt;
   float data;
} hashelement;

HashTable hashtable;
hashtable = DXCreateHash(sizeof(element), NULL, NULL);

for (i=0; i < number of points to insert; i++){
    element.pseudokey = GetKey(&current_point);
    element.pt = current_point;
    element.data = current_data;

    DXInsertHashElement(hashtable, (Element)&element);
}
```

(2) If GetKey returns the same pseudokey for two different points, the second will overwrite the first because no compare function was provided to **DXCreateHash()**.

To extract elements from the hash table:

```
    PseudoKey pkey;
    hashelement *element_ptr;

    pkey = GetKey(&point_to_search_for);
    element_ptr = DXQueryHashElement(hashtable, (Key)&pkey);
```

GetKey that returns a pseudokey given a point x, y, z:

```
PseudoKey GetKey(Key key)
{
  Point *pt;

  pt = (Point *)key;
  return pt->x + 17*pt->y + 23*pt->z;
}
```

Alternatively, the hash function GetKey can be provided at the time the hash table is created. In that case the pseudokey does not need to be part of the element.

```
typedef struct
{
   Point pt;
   float data;
} hashelement;

HashTable hashtable;
hashelement element;

hashtable = DXCreateHash(sizeof(element), GetKey, NULL);

for (i=0; i < number_of_points_to_insert; i++){
    element.pt = current_point;
    element.data = current_data;
    DXInsertHashElement(hashtable, (Element)&element);
```

where:

```
PseudoKey GetKey(Key key)
{
  Point *pt;

  pt = (Point *)key;
  return pt->x + 17*pt->y + 23*pt->z;
}
```

To extract elements from the hash table:

```
hashelement *element_ptr;

element.pt = point_to_search_for;
element_ptr = DXQueryHashElement(hashtable, (Key)&element);
```

(3) This example uses a compare function.

```
typedef struct
 {
   Point pt;
   float data;
 } hashelement;

 HashTable hashtable;
 hashelement element;

 hashtable = DXCreateHash(sizeof(element), GetKey, CompareFunc);
```

```
for (i=0; i < number of points to insert; i++){
    element.pt = current_point;
    element.data = current_data;
    DXInsertHashElement(hashtable, (Element)&element);
 }
```

where the compare function may be:

```
int CompareFunc(Key searchkey, Element element)
{
  Point *p, p1, p2;
  hashelement *h;

  p = (Point *)searchkey;
  p1 = *p;
  h  = (hashelement *)element;
  p2 = h->pt;
  if ((p1.x==p2.x)&&(p1.y==p2.y)&&(p1.z==p2.z))
    return 0;
  else
    return 1;
}
```

13.6 Pick-Assistance Routines

The pick structure output of the Pick tool is a Field and the picked points are listed in the "positions" component of that Field. Other information may be accessed using the pick-assistance routines listed here. These allow you to query the pick structure and to traverse a picked Object. (See 8.2, "ShowPick Module Example—Using Color to Show a Picked Object" on page 56.)

It is recommended that you use the pick-assistance routines to manipulate picked Objects and pick structures, since the pick structure is undefined internally and may change in future.

Error DXGetPickPoint()
Returns the pick point in world coordinates. See page 264.

Error DXQueryPickCount()
Returns the number of picks resulting from a poke. See "Example" on page 143. See page 331.

Error DXQueryPickPath()
Returns information about the pick path. See "Example" on page 143. See page 332.

Error DXQueryPokeCount()
Returns the number of pokes. See "Example" on page 143. See page 332.

Error DXTraversePickPath()
Returns the subObject of the current Object selected by a pick path. See "Example" on page 143. See page 367.

Example

The following code segment finds every picked vertex. (The comment at bottom is left as an exercise for the user.)

```
DXQueryPokeCount(pickField, &nPokes);

for (poke = 0; poke < nPokes; poke++)
{
    DXQueryPickCount(pickField, poke, &nPicks);

    for (pick = 0; pick < nPicks; pick++)
    {
        DXQueryPickPath(pickField, poke, pick,
                        &pathLen, &path, &elementId, &vertexId);

        current = dataObject;
        matrix  = Identity;

        for (i = 0; i < pathLen; i++)
        {
            current = DXTraversePickPath(current, path[i], &matrix);
            if (current == NULL)
                goto error;
        }


     /* now manipulate vertex #vertexId in field current. */

    }
}
```

Chapter 14. Geometric Objects

The routines listed here create geometric Objects such as ribbons, tubes, glyphs, backgrounds, and geometric text. They are implemented by creating Field or Group Objects that can then be rendered.

For detailed descriptions of these routines, see Appendix C, “Data Explorer Library Routines” on page 183.

14.1 Text

Data Explorer supports two varieties of text: geometric and annotation. Geometric text is implemented by stroke and area fonts that can be arbitrarily rotated and scaled before rendering. Annotation text is geometric text that has been transformed so that it always faces the screen. It is supported by a combination of a text routine listed below and the Screen Object described in Chapter 15, “Rendering” on page 149.

`Object DXGetFont()`
Returns a Group containing the specified font. See page 251.

`Object DXGeometricText()`
Produces an Object consisting of the given String. See page 236.

14.2 Clipping

Two higher-level routines that use the Render module’s clipping capability are listed here. These routines do not themselves clip an Object to a plane or to a box, but rather construct an Object (Clipped Object) that describes to the renderer what clipping is to be done. (The renderer does not support nested clipping, and all translucent objects in a scene must be clipped by the same clipping Object.)

`Object DXClipPlane()`
Creates a clipping Object defined by a clipping plane. See page 208.

`Object DXClipBox()`
Creates a clipping Object defined by a clipping box. See page 206.

14.3 Path Operations

The following operations produce a geometric Object from a path. In addition to the functions noted here, the Render module is capable of directly rendering a path as a series of one-pixel lines. A path is a Field with 1-dimensional regular connections. A path can be created by, for example:

```
f = DXNewField();
DXSetComponentValue(f, "positions", ...);
DXSetConnections(f, "lines", DXMakeGridConnections(1, n));
DXEndField(f);
```

where **n** is the number of points.

Both of the operations listed here use “normals” and “tangent” components if they are present; otherwise, they compute approximations to the normals and tangents, as follows: the tangent is the first derivative of the path; the normal is perpendicular to the tangent and lies in the plane formed by the tangent and the second derivative of the path. In each case, appropriate normals are associated with the result for shading.

Object DXRibbon()

Produces a ribbon of the given width from a path or group of paths. See page 340.

Object DXTube()

Produces a tube of a given diameter from a path or group of paths. See page 368.

Chapter 15. Rendering

This chapter describes the Data Explorer rendering model, introduces additional elements of the data model relevant only to rendering, and describes routines for manipulating data structures for rendering. (For descriptions of these routines, see Appendix C, “Data Explorer Library Routines” on page 183.)

The Data Explorer renderer is designed for data visualization. For example, it directly renders scenes described by the Data Explorer data model (see Chapter 3, “Understanding the Data Model” on page 15). The renderer handles all combinations of Groups and Fields as input Objects. The members of a Group or of a subclass of a Group (e.g., Series and Composite Fields) are combined into one image by the renderer.

Rendering a scene involves fours steps:

1. Transformation to *world coordinates*—Applying transforms specified by transform nodes in the Object.
2. Shading—Assigning colors to the vertices, using the intrinsic surface colors, surface normals, surface properties specified by Field components, and lights specified by Light Objects.
3. Transformation to *image coordinates*—Applying transforms specified by a camera Object.
4. Tiling—Generating an image by interpolating point colors and opacities across faces, and rendering volumes with a rendering algorithm.

15.1 Transformation

Transformation is the process of computing pixel coordinates from model coordinates (i.e., the coordinates of the Object). Because **DXRender()** performs necessary transformations, most applications do not need **DXTransform()** .

Transformation is essentially a two-step operation: (1) transform the model coordinates of an Object into world coordinates; and (2) transform the world coordinates into image coordinates (see Figure 10).

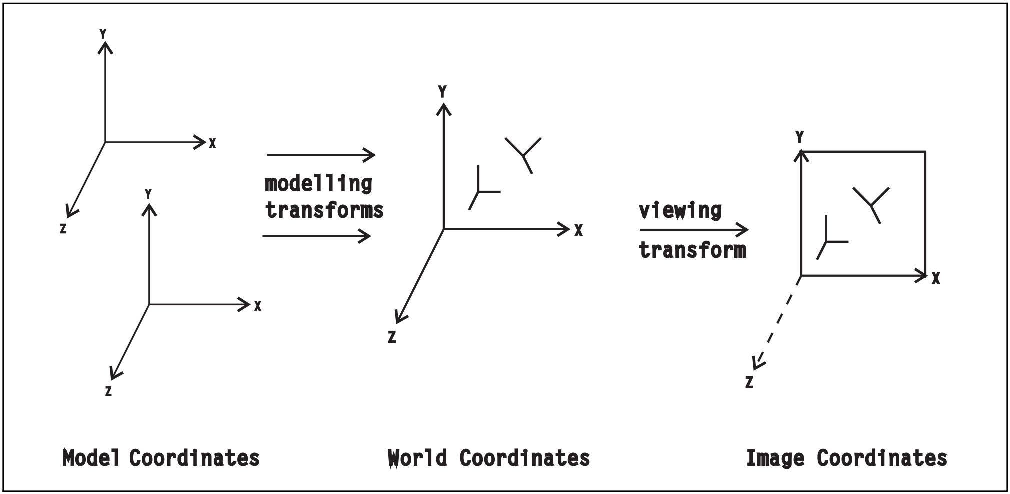


Figure 10. Transformation of Coordinates

The transformation from model to world coordinates is specified by transform nodes (see 15.4, “Xform Class” on page 154) in the description of the input Object. The transformation from world coordinates to image coordinates is specified by a camera Object.

`Object DXApplyTransform()`
Recursively applies a transform to an Object. See page 198.

15.2 Surface Shading

Shading is the process of applying lights to a surface according to shading parameters specified for the surface and the scene.

The shading process described here is performed by the **`DXRender()`** function for surface objects only; volumes are rendered directly, using the colors and opacities specified. Lights are specified by light Objects (see 15.8, "Light Class" on page 156) contained in the input Object. Shading is defined only for 2-dimensional connections (lines, triangles, and quads) and is applied only if normals are present. The shading process uses the following Field components:

Component	Meaning
"positions"	points
"colors"	front and back colors
"front colors"	colors of front of face
"back colors"	colors of back of face
"normals"	surface normals

A Field may have both "colors" and "front colors" or both "colors" and "back colors," in which case the "front colors" or "back colors" component overrides the "colors" component for the specified side of the object. The front and back of a surface are defined in Chapter 3, "Understanding the Data Model" on page 15 in *IBM Visualization Data Explorer User's Guide*.

Shading parameters are specified by a set of attributes of an input Object:

Attribute	Meaning
"ambient"	ambient lighting coefficient k_a
"diffuse"	diffuse lighting coefficient k_d
"specular"	specular lighting coefficient k_s
"shininess"	specular lighting exponent *sp*

These parameters apply to both the front and back of an object. In addition, for each shading parameter "*x*," there is also a "front *x*" and a "back *x*" parameter that apply only to the front and back of a surface respectively. These parameters are used in the following shading model:

$$I = k_a AC + k_d LC\,(\mathbf{n} \cdot \mathbf{l}) + k_s L\,(\mathbf{n} \cdot \mathbf{h})^{sp}$$

where:

I = apparent intensity of the object	k = a lighting coefficient
A = an ambient light	$\mathbf{n}$ = the surface normal
C = the color of the object	$\mathbf{l}$ = the direction to the light
L = a point or distant light	sp = specular lighting exponent

$\mathbf{h}$ = a unit vector halfway between the direction to the camera and the direction to the light.

Color, Opacity, and Normals Dependencies

Colors, opacities, and normals may be dependent on the positions (when the corresponding components have a “dep” attribute of “positions”) or on connections or faces (when the corresponding components have a “dep” attribute of “connections” or “faces”).

Opacities and normals, if present, must depend on the same component that the colors depend upon, with one exception: if the colors are dependent on the positions and the normals are dependent on the connections, the face will be flat-shaded with the average color of the face vertices.

If the colors, opacities, and normals are dependent on the positions, the color and opacity of each face is linearly interpolated between the vertices (Gouraud shading). If they are dependent on the connections or faces, the color and opacity of each face is constant (flat shading).

15.3 Tiling

Tiling is the process of combining shaded surface and volume interpolation elements to produce an image. The following table lists elements that are defined and implemented (“yes”). A dash identifies a meaningless combination.

Component	**irregular**	**regular**	**opaque**	**translucent**
“lines”	yes	yes	yes	yes
“triangles”	yes	—	yes	yes
“quads”	yes	yes	yes	yes
“tetrahedra”	yes	—	—	yes
“cubes”	yes	yes	—	yes
“faces,” “loops,” “edges”	yes	—	yes	yes

Lines may be irregular unconnected vectors or *paths* having regular 1-dimensional connections. Surfaces and volumes may be (1) completely irregular, (2) regular in connections but irregular in positions, or (3) regular in both connections and positions (Figure 11 on page 153).

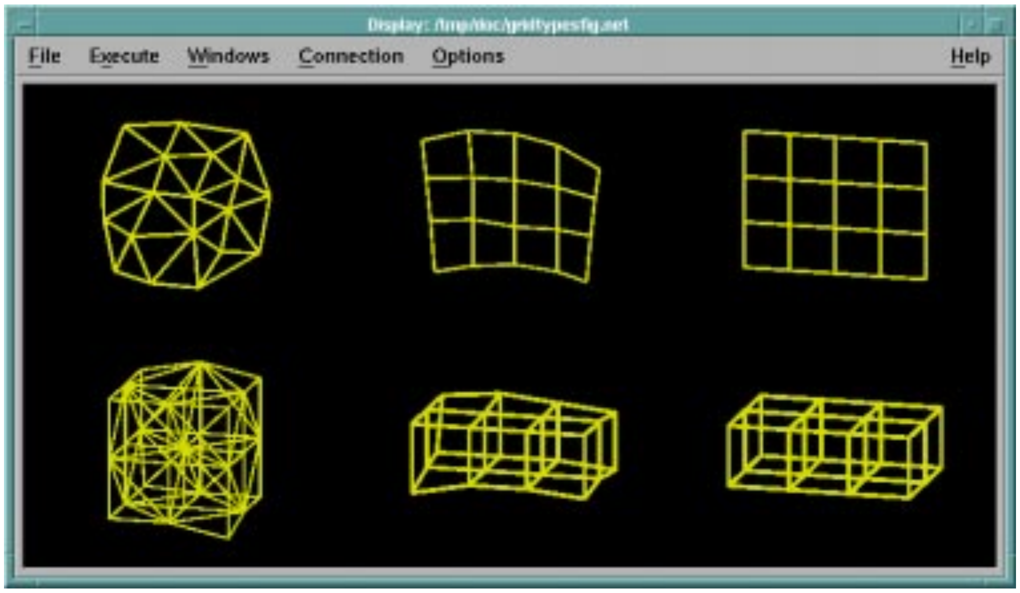


Figure 11. Types of Connections and Positions

Rendering Model

The interpretation of “colors” and “opacities” differs between surfaces and volumes. For surfaces, a surface of color c_f and opacity o are combined with the color c_b of the objects behind it, resulting in a combined color $c_f o + c_b (1 - o)$.

For volumes, the “dense emitter” model is used, in which the opacity represents the instantaneous rate of absorption of light passing through the volume per unit thickness, and the color represents the instantaneous rate of light emission per unit thickness. If $c(z)$ represents the color of the object at z and $o(z)$ represents its opacity at z, the total color c of a ray passing through the volume is given by:

$$c = \int_{-\infty}^{\infty} c(z) \exp\left(-\int_{-\infty}^{z} o(\zeta) d\zeta \right) dz$$

Tiling Options

Tiling options are controlled by a set of Object attributes. These attributes may be associated with Objects at any level of a Field/Group hierarchy. The attributes may be set by using the **`DXSetAttribute()`** function, or by using the Options module. The attributes are:

Attribute	Meaning
“fuzz”	object fuzz

Object fuzz is a method of resolving conflicts between objects at the same distance from the camera. For example, it may be desirable to define a set of lines coincident with a plane. Normally it will be unclear which object is to be displayed in front. In addition, single-pixel lines are inherently inaccurate (i.e. they deviate from the actual geometric line) by as much as one-half pixel; when displayed against a sloping surface, this x or y inaccuracy is equivalent to a z inaccuracy related to the slope of the surface. The “fuzz” attribute specifies a z value that will be added to the object before it is compared with other objects in the scene, thus resolving this problem. The fuzz value is specified in pixels. For example, a fuzz value of one pixel can compensate for the described half-pixel inaccuracy when the line is displayed against a surface with a slope of two.

15.4 Xform Class

An Xform, or transform Object, is a renderable Object that specifies a modeling transformation matrix applied to another renderable Object. It is included in Groups that are input to the renderer to specify the transformation from model coordinates to world coordinates. See the description of the **DXApplyTransform()** routine in 15.1, "Transformation" on page 150.

Xform DXNewXform()
Creates a new transform Object. See page 313.

Xform DXGetXformInfo()
Extracts information from a transform Object. See page 272.

Xform DXSetXformObject()
Sets the Object to which a transform is applied. See page 363.

15.5 Screen Class

A Screen Object is an Object that maintains a size and alignment with the screen (output image) independent of the camera view and scaling transformations applied to it.

Three options are provided for the interpretation of translations applied to a Screen Object. First, a translation applied to the Screen Object may specify a new position for the origin of the Screen Object in world space (**SCREEN_WORLD**). Second, a translation applied to the Screen Object may specify a new location for the Screen Object in the image, measured in pixels, where (0,0) refers to the lower-left corner of the image (**SCREEN_PIXEL**). Third, a translation applied to the Screen Object may specify a new location for the Screen Object in the image, measured in viewport-relative coordinates, where (0,0) refers to the lower-left corner of the image and (1,1) refers to the upper-right corner of the image (**SCREEN_VIEWPORT**).

In addition, with regard to *z*, the object may be displayed either in place in the scene, in front of all objects, or behind all objects, according to whether the **z** parameter to **DXNewScreen()** is 0, *+1*, or *-1* respectively.

```
#define SCREEN_WORLD 0
#define SCREEN_VIEWPORT 1
#define SCREEN_PIXEL 2
```

Screen DXNewScreen()
Creates a new Object aligned to the final screen. See page 311.

Screen DXGetScreenInfo()
Returns information about a Screen Object. See page 267.

Screen DXSetScreenObject()
Sets the Object to which a screen transform is to be applied. See page 361.

15.6 Clipped Class

A Clipped Object is one Object clipped by another. The first Object is actually rendered; the second represents a region to which the first Object is clipped. The clipping Object is expected to be a closed convex surface. The portion of the first Object that is within the clipping Object is rendered. If the clipping Object is not a closed convex surface, the results are undefined. The clipping is performed by the renderer during the rendering process. Thus clipping is provided as a data structure for representing the Clipped Object to the renderer, rather than as an explicit operation.

`Clipped DXNewClipped()`
Creates a new Clipped Object. See page 300.

`Clipped DXGetClippedInfo()`
Returns the Object to be rendered and the clipping Object. See page 242.

`Clipped DXSetClippedObjects()`
Sets the Object to be rendered and the Object to clip it with during the rendering process. See page 347.

15.7 Camera Class

A Camera Object stores parameters that relate a scene to an image of the scene, including camera position and orientation, field of view, type of projection, and resolution. It specifies the transformation from world coordinates to image coordinates.

`Camera DXNewCamera()`
Creates a new Camera. See page 299.

`Camera DXSetView()`
Specifies the Camera position, a point on the line of sight of the Camera, and the Camera orientation. See page 363.

`Camera DXSetOrthographic()`
Specifies an orthographic view. The width of the viewport in the world coordinates and the aspect ratio are specified. See page 356.

`Camera DXSetPerspective()`
Specifies a perspective view. The field of view specifies the tangent of half the angle of the field of view. The height of the view is the specified **`aspect`** times the width. See page 359.

`Camera DXSetBackgroundColor()`
Specifies the background color of a scene. See page 345.

`Camera DXSetResolution()`
Specifies the resolution of a Camera. See page 360.

`Camera DXGetView()`
Returns the camera view parameters. See page 363.

`Camera DXGetOrthographic()`
Returns the orthographic camera parameter. See page 356.

`Camera DXGetPerspective()`
Returns the perspective camera parameters. See page 359.

Camera DXGetBackgroundColor()
Extracts a scene background color associated with a camera. See page 345.

Camera DXGetCameraResolution()
Returns the camera resolution. See page 360.

Matrix DXGetCameraMatrix()
Matrix DXGetCameraMatrixWithFuzz();
Matrix DXGetCameraRotation();
Return matrices that represent stages of the viewing operation. See page 241.

15.8 Light Class

Light Objects specify lights for shading. They may be placed in a scene and transformed along with other Objects in the scene (e.g, by changing their positions).

Light DXNewDistantLight()
Creates a distant Light Object. See page 302.

Light DXQueryDistantLight()
Returns information about a Distant Light. See page 326.

Light DXNewAmbientLight()
Creates a Light Object representing an ambient light source. See page 297.

Light DXQueryAmbientLight()
Returns the color of an Ambient Light. See page 323.

15.9 Image Fields

An image Field is a Field with regular positions and with regular quadrilateral connections. Images also contain a “colors” component. The sign of the deltas of the “positions” component determines the orientation of the image.

Functions of the routines listed here include creating a field image, returning a pointer to the data in an image field, and returning information about an image.

Field DXMakeImage()
Creates a new empty image Field. See page 291.

RGBColor *DXGetPixels()
Returns a pointer to the data in an image Field. See page 264.

Field DXGetImageSize()
Object DXGetImageBounds();
Return information about image Fields. See page 252.

Field DXOutputRGB()
Writes an image to a file in RGB format. See page 315.

Object DXDisplayX()
Object DXDisplayX8();
Object DXDisplayX12();
Object DXDisplayX24();
Display an image on an X window. See page 222.

Chapter 16. DXLink Developer's Toolkit

16.1 Introduction

DXLink is a C programming interface that can be used to communicate with the Data Explorer user interface (dxui) or the Data Explorer executive (dxexec). Functions are provided to load programs, enable the setting and retrieval of named variables, control execution, handle errors, and define application-specific messaging. For the user interface, functions are also available to control window visibility and to load configuration files. For the executive, a function is provided to send arbitrary scripting-language commands. Support for X windows is built in but is not required.

The kinds of application that can be written with DXLink include:

- A graphical user interface and demonstration utility that controls the execution of the Data Explorer user interface to start selected demos, flip through images, and remove and place windows on the display.
- A graphical user interface that communicates with the executive, replacing the Data Explorer user interface.
- A shell-like scripting language to control the user interface (and that might also be useful for a demonstration utility).

A majority of DXLink functions have names that begin with the prefix “DXL.” These functions can be used in communicating with either the user interface or the executive. The prefix “uiDXL” identifies a function intended for use only with the interface; the prefix “exDXL,” a function for use only with the executive.

Note: In the remainder of this document, references to “the server” apply to the user interface and the executive equally.

All applications that use the DXLink facilities must link with the library libDXL.a and include the header file dxl.h in the source code.

The functions provided by DXLink are described in more detail in the following sections. They have been divided into four groups:

1. 16.5, “Initialization and Exit” on page 169,
2. 16.6, “Messaging System” on page 170,,
3. 16.7, “Execution Control” on page 173, and
4. 16.8, “Program Control” on page 174.

The declarations for these functions are found in the C include file `dxl.h`, which should be included in any C file that uses them. The following sections describe three simple DXLink programs. All of these examples and necessary Makefiles can be found in `/usr/lpp/dx/samples/dxlink`. It is recommended that you create these programs and run them before studying the C code which follows. See the `Readme` file in `/usr/lpp/dx/samples/dxlink` for instructions on how to create them.

Stand-alone Programs

1. Stand-alone programs may access the Data Explorer data model by linking to the library DXlite. The file libDXlite.a contains a subset of Data Explorer routines (see Appendix B, “Data Explorer Data Model Library: DXlite Routines” on page 181).
2. Stand-alone programs may also access almost all Data Explorer data modules and all Data Explorer library routines by linking to the library libDXcallm.a. (The exceptions are such user-interface features as interactors, the Colormap Editor, the Image tool, and Get and Set.) The file libDXcallm.a contains all the Data Explorer library routines listed in Appendix C, “Data Explorer Library Routines” on page 183. See also 12.10, “Module Access” on page 127.
3. When starting Data Explorer from an external program, certain command line options may be useful to disable portions of the user interface that the external program is intended to control. See Table 6 on page 297 in *IBM Visualization Data Explorer User’s Guide*.

16.2 Example 1: sealevel.c

The first example which we will discuss is sealevel.c. This sample program starts the Data Explorer user interface in **-image** mode, and then loads a visual program (see Figure 12).

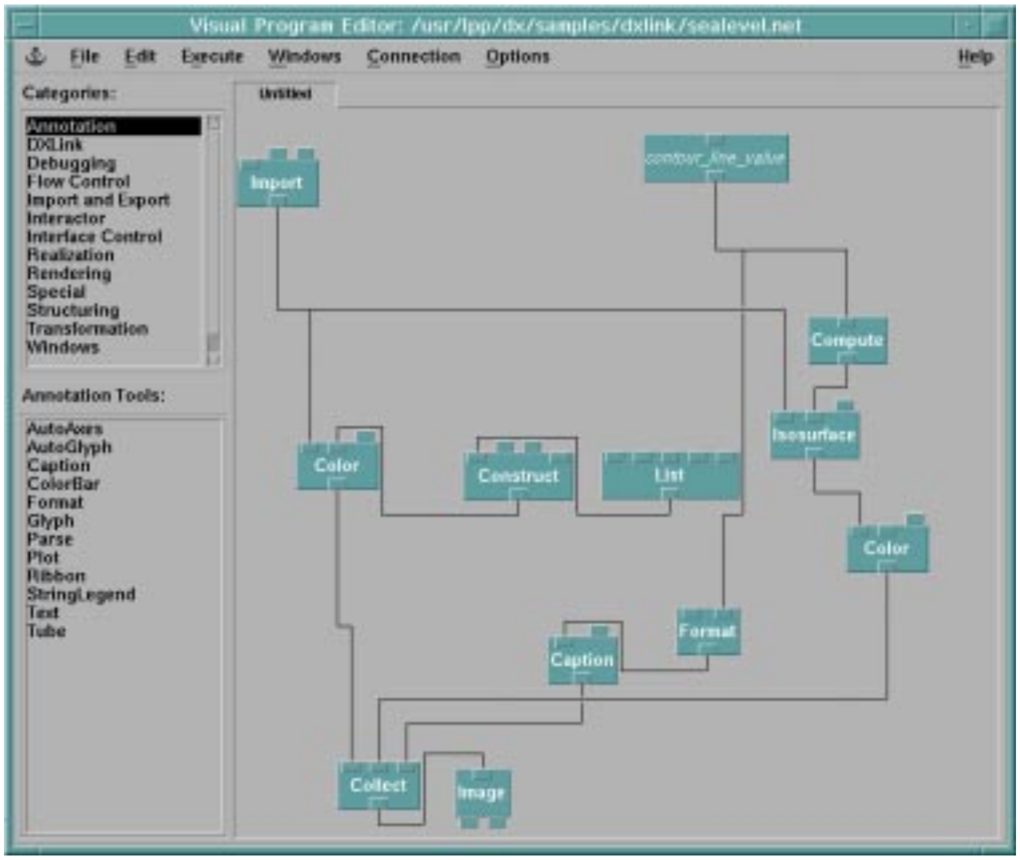


Figure 12. sealevel.net

The visual program contains a DXLInput tool which can receive values from the DXLink program. It is named **contour_line_value**. The DXLink program sends several different values to this DXLInput tool, and the resulting image is displayed to the user.

```
#include <stdio.h>
#include "dx/dxl.h"

#ifndef BASE
#define BASE "/usr/lpp/dx"
#endif


/*
 * define an error handler
 */
void ErrorHandler(DXLConnection *conn, const char *msg, void *data)
{
    printf("DXL Error: %s\n", msg);
}



main(int argc, char *argv[])
{
    DXLConnection *conn = NULL;
    char result[100];


    /*
     * Start Data Explorer in -image mode with certain menus disabled.
     */
    conn = DXLStartDX(
              "dx -image -noExitOptions -noExecuteMenus -noConnectionMenus",
              NULL);

    if (conn == NULL)
    {
        fprintf(stderr,"Could not connect\n");
        perror("DXLStartDX");
        exit(1);
    }


    /*
     * Set the error handler
     */
    DXLSetErrorHandler(conn, ErrorHandler, NULL);


    /*
     * Load the visual program to run
     */
    DXLLoadVisualProgram(conn, BASE"/samples/dxlink/sealevel.net");

    /*
     * Set the value of the DXLInput tool which is labelled
```

```
         * "contour_line_value" and execute.
         */
        DXLSetValue(conn, "contour_line_value", "0");
        DXLExecuteOnce(conn);


        /*
         * Set the value of the DXLInput tool which is labelled
         * "contour_line_value" and execute.
         */
        DXLSetValue(conn, "contour_line_value", "2");
        DXLExecuteOnce(conn);

        /*
         * Set the value of the DXLInput tool which is labelled
         * "contour_line_value" and execute.
         */
        DXLSetValue(conn, "contour_line_value", "5");
        DXLExecuteOnce(conn);

        /*
         * Set the value of the DXLInput tool which is labelled
         * "contour_line_value" and execute.
         */
        DXLSetValue(conn, "contour_line_value", "20");
        DXLExecuteOnce(conn);

        /*
         * Set the value of the DXLInput tool which is labelled
         * "contour_line_value" and execute.
         */
        DXLSetValue(conn, "contour_line_value", "50");
        DXLExecuteOnce(conn);

        printf("An image window will appear\n");
        printf("and a sequence of images will be created.\n");
        printf("When you are finished, hit return to quit:");
        gets(result);
        DXLExitDX(conn);
    }
```

16.3 Example 2: maptoplane.c

The second example which we will discuss is maptoplane.c. This sample program starts the Data Explorer user interface in **-edit** mode, and then loads a visual program (see Figure 13 on page 162).

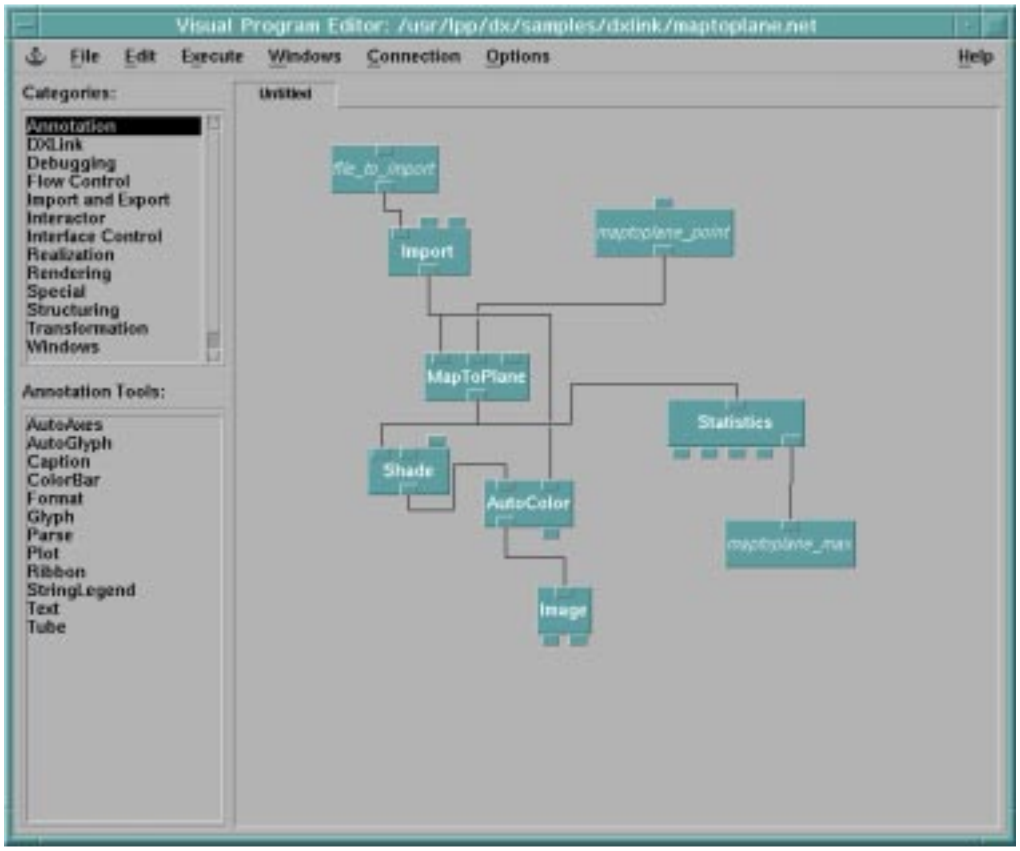


Figure 13. maptoplane.net

The visual program contains two DXLInput tool which can receive values from the DXLink program. One is named **`file_to_import`**, and the other is named **`maptoplane_point`**. The DXLink program sends the filename to **`file_to_import`**, and then sends several different values to **`maptoplane_point`**. The program is run and for each execution, statistics are computed on the resulting MapToPlane. The maximum value on the plane is passed to a DXLOutput tool labeled **`maptoplane_max`**. In maptoplane.c, a handler has been installed for output coming from **`maptoplane_max`**, and the handler simply prints the value to the terminal.

```
#include <stdio.h>
#include "dx/dxl.h"

#ifndef BASE
#define BASE "/usr/lpp/dx"
#endif


void SyncAfterExecute(DXLConnection *conn)
{
  int status=1;

   while (status) {
      sleep(1);
      if (DXLIsMessagePending(conn))
         DXLHandlePendingMessages(conn);
      DXLGetExecutionStatus(conn, &status);
   }
}


/*
 * this routine simply prints the maximum value as received from the
 * DXLOutput tool
 */
void max_handler(DXLConnection *conn, const char *name, const char *value,
                 void *data)
{
   printf("max value = %s\n", value);
}



main(int argc, char *argv[])
{
    DXLConnection *conn = NULL;
    char result[100];
    int status;

    /*
     * Start Data explorer user interface in -edit mode, with certain
     * in -edit mode, with certain menus turned off.
     */
    conn = DXLStartDX("dx -edit -noExitOptions -noExecuteMenus -noConnectionMenus",
                      NULL);


    if (conn == NULL)
    {
        fprintf(stderr,"Could not connect\n");
        perror("DXLStartDX");
        exit(1);
    }

    /*
     *  Set the handler for the DXLOutput tool which is labelled
     *  "maptoplane_max"
     */
    DXLSetValueHandler(conn, "maptoplane_max", max_handler, NULL);


    /*
     * Load the visual program to run. Set the value of the DXLInput
     * tool which is labelled "file_to_import".
     * Also set the value of the DXLInput tool which is labelled
     * "maptoplane_point".
     */
    DXLLoadVisualProgram(conn, BASE"/samples/dxlink/maptoplane.net");
    DXLSetString(conn, "file_to_import","/usr/lpp/dx/samples/data/temperature");
    DXLSetValue(conn, "maptoplane_point", "[0 5000 5000]");

    /*
     * Execute the visual program and check for input from maptoplane_max.
     */
    DXLExecuteOnce(conn);
    SyncAfterExecute(conn);

    /* Change the value for the DXLInput tool labelled "maptoplane_point
     * and execute again.
     */
    DXLSetValue(conn, "maptoplane_point", "[10000 5000 5000]");
    DXLExecuteOnce(conn);
    SyncAfterExecute(conn);

    /* Change the value for the DXLInput tool labelled "maptoplane_point
     * and execute again.
     */
    DXLSetValue(conn, "maptoplane_point", "[20000 5000 5000]");
    DXLExecuteOnce(conn);
    SyncAfterExecute(conn);

    /* Change the value for the DXLInput tool labelled "maptoplane_point
     * and execute again.
     */
    DXLSetValue(conn, "maptoplane_point", "[30000 5000 5000]");
    DXLExecuteOnce(conn);
    SyncAfterExecute(conn);

    /* Change the value for the DXLInput tool labelled "maptoplane_point
     * and execute again.
     */
    DXLSetValue(conn, "maptoplane_point", "[50000 5000 5000]");
    DXLExecuteOnce(conn);
    SyncAfterExecute(conn);

    printf("Hit return to quit:");
    gets(result);
    DXLExitDX(conn);
}
```

16.4 Example 3: xapp.c

The third example which we will discuss is xapp.c. This sample program does not use the Data Explorer user interface at all; rather it creates its own (simple) user interface. The program communicates with Data Explorer entirely through the scripting language.

The interface presents the user with four buttons. Depending on which button is pressed, a different data file is imported. The maximum value in the data set is then computed and sent back to the DXLink application using a DXLOutput tool. The result is then displayed in a text widget created by the DXLink program.

```
#include <Xm/Xm.h>
#include <Xm/Form.h>
#include <Xm/Label.h>
#include <Xm/PushB.h>
#include <Xm/ToggleB.h>
#include <Xm/Text.h>
#include "dx/dxl.h"


void radio_cloudCB(Widget, XtPointer, XtPointer);
void radio_rainCB(Widget, XtPointer, XtPointer);
void radio_windCB(Widget, XtPointer, XtPointer);
void radio_tempCB(Widget, XtPointer, XtPointer);
void cloudhandler(DXLConnection *, const char *, const char *, void *);
void rainhandler(DXLConnection *, const char *, const char *, void *);
void windhandler(DXLConnection *, const char *, const char *, void *);
void temphandler(DXLConnection *, const char *, const char *, void *);

static String DefaultResources[] =
{
    "*background:             #b4b4b4b4b4b4",
    "*foreground:             black",
#ifdef sgi
    "*fontList:                -adobe-helvetica*bold-r*14*=bold\n\
                               -adobe-helvetica*medium-r*14*=normal\n\
                               -adobe-helvetica*medium-o*14*=oblique",
#else
    "*fontList:                -adobe-helvetica*bold-r*14*=bold\
                               -adobe-helvetica*medium-r*14*=normal\
                               -adobe-helvetica*medium-o*14*=oblique",
#endif
   "*XmToggleButton.selectColor:            CadetBlue",
    "*XmText.shadowThickness:                1",
    NULL
};


main(argc, argv)
int argc;
char *argv[];
{
   Widget          toplevel, main_w, label, textfield, radio_box;
   Widget          radio_cloud, radio_rain, radio_temp, radio_wind;
   XtAppContext    app;
   XmString        xms;
   DXLConnection   *conn;
   int             n;
   Arg             wargs[50];


   /*
    *  Start the Data Explorer executive.
    */
   conn = DXLStartDX("dx -exonly",NULL);
   if (!conn)
    {
      printf("could not start dx");
      exit(0);
```

```
 }


XtSetLanguageProc (NULL, NULL, NULL);

toplevel = XtVaAppInitialize (&app, "Demos",
                              NULL, 0, &argc, argv,
                              DefaultResources, NULL);

DXLInitializeXMainLoop(app, conn);


/*
 * Create the user interface for this application
 */
main_w = XtVaCreateManagedWidget("form",
              xmFormWidgetClass, toplevel,
              XmNwidth,    400,
              XmNheight,   180,
              XmNfractionBase, 5,
              NULL);

n = 0;
XtSetArg(wargs[n], XmNtopAttachment, XmATTACH_FORM); n++;
XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
XtSetArg(wargs[n], XmNleftOffset, 30); n++;
radio_box = (Widget)XmCreateRadioBox(main_w, "choice", wargs, n);
XtManageChild(radio_box);

xms = XmStringCreateSimple("cloudwater");
radio_cloud = XtVaCreateManagedWidget("radio_cloud",
                   xmToggleButtonWidgetClass, radio_box,
                   XmNlabelString, xms,
                   NULL);
XmStringFree(xms);
XtAddCallback(radio_cloud, XmNvalueChangedCallback,
              (XtCallbackProc)radio_cloudCB,
              (XtPointer)conn);

xms = XmStringCreateSimple("rainwater");
radio_rain = XtVaCreateManagedWidget("radio_rain",
                   xmToggleButtonWidgetClass, radio_box,
                   XmNlabelString, xms,
                   NULL);
XmStringFree(xms);
XtAddCallback(radio_rain, XmNvalueChangedCallback,
              (XtCallbackProc)radio_rainCB,
              (XtPointer)conn);

xms = XmStringCreateSimple("temperature");
radio_temp = XtVaCreateManagedWidget("radio_temp",
                   xmToggleButtonWidgetClass, radio_box,
                   XmNlabelString, xms,
                   NULL);
XmStringFree(xms);
XtAddCallback(radio_temp, XmNvalueChangedCallback,
              (XtCallbackProc)radio_tempCB,
              (XtPointer)conn);
```

```
   xms = XmStringCreateSimple("wind");
   radio_wind = XtVaCreateManagedWidget("radio_wind",
                   xmToggleButtonWidgetClass, radio_box,
                   XmNlabelString, xms,
                   NULL);
   XmStringFree(xms);
   XtAddCallback(radio_wind, XmNvalueChangedCallback,
              (XtCallbackProc)radio_windCB,
              (XtPointer)conn);




   xms = XmStringCreateSimple("returned value:");
   label = XtVaCreateManagedWidget("label",
              xmLabelWidgetClass,
              main_w,
              XmNtopAttachment,   XmATTACH_WIDGET,
              XmNtopWidget,       radio_box,
              XmNbottomAttachment,   XmATTACH_FORM,
              XmNleftAttachment,  XmATTACH_FORM,
              XmNlabelString, xms,
              NULL);
   XmStringFree(xms);

   textfield = XtVaCreateManagedWidget("text",
              xmTextWidgetClass,
              main_w,
              XmNtopAttachment,   XmATTACH_OPPOSITE_WIDGET,
              XmNtopWidget,       label,
              XmNleftAttachment,  XmATTACH_WIDGET,
              XmNleftWidget,      label,
              XmNrightAttachment,  XmATTACH_FORM,
              XmNbottomAttachment,  XmATTACH_FORM,
              NULL);


   /*
    * Set the handlers for the various parameters
    */
   DXLSetValueHandler(conn,"cloudmax", cloudhandler, textfield);
   DXLSetValueHandler(conn,"rainmax", rainhandler, textfield);
   DXLSetValueHandler(conn,"windmax", windhandler, textfield);
   DXLSetValueHandler(conn,"tempmax", temphandler, textfield);


   XtRealizeWidget (toplevel);
   XtAppMainLoop (app);
}


/*
 * The following are the handlers for data coming from DXLOutput. If
 * data is received by the handler, it is presented in a text widget.
 */
void rainhandler(DXLConnection *conn, const char *name, const char *value,
             void *data)
{
   char string[100];
   Widget text_widget = (Widget)data;

   sprintf(string,"rainwater max value = %s", value);
   XmTextSetString(text_widget, string);
}
void cloudhandler(DXLConnection *conn, const char *name, const char *value,
             void *data)
{
   char string[100];
   Widget text_widget = (Widget)data;

   sprintf(string,"cloudwater max value = %s", value);
   XmTextSetString(text_widget, string);
}
void temphandler(DXLConnection *conn, const char *name, const char *value,
             void *data)
{
   char string[100];
   Widget text_widget = (Widget)data;

   sprintf(string,"temperature max value = %s", value);
   XmTextSetString(text_widget, string);
}
void windhandler(DXLConnection *conn, const char *name, const char *value,
             void *data)
{
   char string[100];
   Widget text_widget = (Widget)data;

   sprintf(string,"wind max value = %s", value);
   XmTextSetString(text_widget, string);
}


/*
 * The following are the callbacks for the buttons in the
 * user interface created above. In each case, some simple
 * script language commands are sent to the Data Explorer
 * executive. The maximum as computed by the Statistics
 * module is then input to the DXLOutput tool. The handlers
 * defined above wait for values to be received from
 * DXLOutput, and then present the result in the text widget.
 */
void radio_cloudCB(Widget w, XtPointer xp1, XtPointer xp2)
{
   DXLConnection *conn = (DXLConnection *)xp1;
   DXLSend(conn, "g = Import(\"/usr/lpp/dx/samples/data/cloudwater\");");
   DXLSend(conn, "mean,sd,var,min,max = Statistics(g);");
   DXLSend(conn, "DXLOutput(\"cloudmax\", max);");
}
void radio_rainCB(Widget w, XtPointer xp1, XtPointer xp2)
{
   DXLConnection *conn = (DXLConnection *)xp1;
   DXLSend(conn, "g = Import(\"/usr/lpp/dx/samples/data/rainwater\");");
   DXLSend(conn, "mean,sd,var,min,max = Statistics(g);");
   DXLSend(conn, "DXLOutput(\"rainmax\", max);");
}
void radio_tempCB(Widget w, XtPointer xp1, XtPointer xp2)
{
   DXLConnection *conn = (DXLConnection *)xp1;
   DXLSend(conn, "g = Import(\"/usr/lpp/dx/samples/data/temperature\");");
   DXLSend(conn, "mean,sd,var,min,max = Statistics(g);");
   DXLSend(conn, "DXLOutput(\"tempmax\", max);");
}
void radio_windCB(Widget w, XtPointer xp1, XtPointer xp2)
{
   DXLConnection *conn = (DXLConnection *)xp1;
   DXLSend(conn, "g = Import(\"/usr/lpp/dx/samples/data/wind\");");
   DXLSend(conn, "mean,sd,var,min,max = Statistics(g);");
   DXLSend(conn, "DXLOutput(\"windmax\", max);");
}
```

16.5 Initialization and Exit

These routines enable an application to initiate and terminate connections with Data Explorer. DXLStartDX establishes a connection ***conn** for a Data Explorer instance. It is through this connection that information is sent and received during a Data Explorer session.

void DXLCloseConnection(DXLConnection *conn)
Closes the connection to Data Explorer (***conn**) and frees the memory allocated to ***conn** but does not terminate the Data Explorer session.

DXLConnection *DXLConnectToRunningServer(int port, const char *host)
This routine is used primarily for debugging purposes, to connect to a server (i.e., dxexec or dxui) that has already been started externally and is waiting for a connection (see `-exonly` and `-appPort` options). The routine creates a connection between the calling application and Data Explorer at the specified port on the specified host.

The parameter **port** must be greater than or equal to zero. If **host** is specified as NULL, the function uses “localhost”.

DXLError DXLExitDX(DXLConnection *conn)
Closes the connection to Data Explorer (***conn**) and terminates the Data Explorer session. Returns OK or ERROR.

int DXLGetSocket(DXLConnection *conn)
Returns the socket number associated with connection ***conn**.

DXLError DXLInitializeXMainLoop(XtAppContext app, DXLConnection *conn)
Initializes the X11 window system so that calls to XtAppMainLoop will cause messages to be processed. (See 16.6, “Messaging System” on page 170.)

int DXLSetMessageDebugging(DXLConnection *c, int on)
Specifies the on-off status of message debugging. If it is enabled, messages are printed to the terminal window as they are sent or received. The return value identifies the previous state of message debugging.

void DXLSetSynchronization(DXLConnection *conn, const int onoff)
Sets the synchronization status of the connection to Data Explorer. When **onoff** is set to 1, the connection is synchronized; when set to 0, it is not. With the connection synchronized, routines wait for a response from the “server” (i.e., UI or exec) after a message is sent. For example, if

synchronization has been turned on, a call to DXLSetValue will not return until the server has processed the set value.

`DXLConnection *DXLStartDX(const char *string, char *host)`

Starts Data Explorer and creates a connection to it. The parameter **`*string`** is the command that would be used to start Data Explorer at the command line. For example, **`string`** could be:

```
dx -image -mdf user.mdf
```

Note that you must specify either -image, -edit, or -menubar in order to bypass the Data Explorer Startup window. It is not possible to use DXLStartDX to start Data Explorer via the Startup window. The parameter **`*host`** is the name of the host on which Data Explorer is to be run. If specified as NULL, the local host is assumed.

The routine returns a pointer to a DXLConnection on success or it returns NULL and sets an error code (available in the global variable **`errno`**). The connection structure, specified by

```
typedef struct DXLConnection DXLConnection;
```

is the primary structure used by DXLink for maintaining information about the connection to the server. This DXLConnection is passed to most DXLink routines to indicate the relevant server connection.

To start the Data Explorer executive and connect DXLink to it, you might issue the following call:

```
DXLStartDX("dx -cache off -exonly");
```

The `-exonly` flag causes the executive to start up and to wait for a connection from DXLink. It is therefore required when connecting to the executive. Note that the “-script” option should not be used, as it causes the executive to start up in script mode, which requires commands typed directly at a prompt, bypassing the message system.

Similarly, a connection to the Data Explorer user interface can be initiated with the command:

```
DXLStartDX("dx -mdf my.mdf -image");
```

```
void DXLSetBrokenConnectionCallback(DXLConnection *conn,
                void (*proc)(DXLConnection *, void *), void *data);
```

Allows the application to install a routine to be called when the connection to Data Explorer is broken.

16.6 Messaging System

The primary functionality provided by DXLink is the sending of messages to the server and the handling of messages from the server. Messages sent to the server include the setting of variable values and the initiation or termination of execution. Messages sent from the server and handled by the DXLink library include errors, warnings, variable values, and execution state. For the most part, the message format is hidden in the programming interface.

Sending Messages to the Server

The function DXLSend (see “Messaging Routines” on page 172) sends specific messages directly to the server. In general, one needs to be sure that the server will either handle the message properly or ignore it. The user interface will ignore most unrecognized messages, while the executive will accept most legal scripting-language commands.“Setting Variables” on page 175 discusses how an application program can set the value of variables in a visual program.

Other functions may also result in messages to the server. For example, DXLStartDX initiates the connection to the server and manages all the messaging associated with establishing the connection to Data Explorer.

When a function call results in sending a message to the server, DXLink can either synchronize with the server to ensure that the message has been accepted or it can return without receiving an acknowledgment. By default, DXLink is configured to use the latter (asynchronous) method. DXLSetSynchronous sets the synchronization method to be used. DXLSync allows synchronization at specified points in an application that does not use synchronous mode.

Receiving Messages from the Server

DXLink always uses an asynchronous method of handling messages sent from the server to the DXLink application.

DXLHandlePendingMessages must be called when messages are pending and ready to be processed. This function arranges for message handlers to be called for pending messages and it discards any messages that do not have handlers installed. The function DXLIsMessagePending determines whether DXLHandlePendingMessages needs to be called and allows an application to poll the connection to the server for messages that need to be processed.

The function DXLGetSocket is provided for systems with socket support. The returned socket can be used to arrange for the operating system to perform a blocking **`select( )`** command on the socket to determine when there are messages available. This might be used in a scripting application that uses **`select( )`** on both the DXLink socket and the file descriptor corresponding to the input device.

The DXLOutput module can be used to send Data Explorer values from Data Explorer to a DXLink application. For descriptions of both routines, see *IBM Visualization Data Explorer User’s Reference*. For a discussion of Data Explorer values, see “Setting Variables” on page 175.

X11 Windows: The function DXLInitForXMainLoop provides support for applications built under the X window system. This function should be called before entering the X main event loop; it arranges for DXLHandlePendingMessages to be called automatically when messages are available. See 16.5, “Initialization and Exit” on page 169.

Message handlers are called from DXLHandlePendingMessages when the indicated message is encountered. DXLink installs a number of its own message handlers.

The message handler structure is specified by:

```
typedef void (*DXLMessageHandler)(DXLConnection *conn, const char *msg, void *data);
```

The packet types for messages are defined as follows:

```
enum DXLPacketType {
    PACK_INTERRUPT      = 1,
    PACK_MACRODEF       = 4,
    PACK_FOREGROUND     = 5,
    PACK_BACKGROUND     = 6,
    PACK_ERROR          = 7,
    PACK_MESSAGE        = 8,
    PACK_INFO           = 9,
    PACK_LINQUIRY       = 10,
    PACK_LRESPONSE      = 11,
    PACK_COMPLETE       = 19,
    PACK_LINK           = 22
};
typedef enum DXLPacketType DXLPacketTypeEnum;
```

Messaging Routines

```
DXLError DXLSetMessageHandler(DXLConnection *conn,
                      DXLPacketTypeEnum type, const char *matchstr,
                      DXLMessageHandler h, const void *data);
```

Sets a message handler. This routine allows the user to install a message handler for messages of any type. The handler “h” will be called, receiving the pointer “data”, whenever the message handling infrastructure receives a message of type “type” containing a message that matches “matchstr”.

```
DXLError DXLRemoveMessageHandler(DXLConnection *conn,
                      DXLPacketTypeEnum type, const char *matchstr,
                      DXLMessageHandler h);
```

Removes a message handler. The “h” argument is ignored.

```
int DXLIsMessagePending(DXLConnection *conn)
```

Can be used by applications that need to poll the DXLConnection to determine whether there are messages from the server that should be processed with DXLHandlePendingMessages. It returns zero (0) if there are no messages to handle, and a nonzero value otherwise.

Note: In windowing applications that use DXLInitializeXMainLoop, this function is not needed.

```
DXLError DXLHandlePendingMessages(DXLConnection *conn)
```

Parses a message that is waiting to be processed. The result is a call to the installed message handlers. This routine is called automatically if DXLInitializeXmainLoop is used.

```
DXLError DXLSend(DXLConnection *conn, const char *msg)
```

When the server is the executive, it will accept most one-line scripting-language commands, including assignments and module calls. For example, the following command makes a compound assignment (1.23 to the variable “foo” and [1 2 3] to the vector “bar”):

```
DXLSend(conn, "foo, bar = 1.23, [1 2 3]; \n");
```

Notes:

1. Multiline scripting-language commands (e.g., macro definitions) cannot be sent with this function. Instead see the two `...MacroDefinition` routines in 16.8, "Program Control" on page 174.
2. When the server is the user interface, it will ignore most commands, and this function should be avoided. Instead see the `DXLSetValue` functions in "Setting Variables" on page 175.

```
DXLError DXLSetErrorHandler(DXLConnection *conn, DXLMessageHandler h,
                                          const void *data);
```

Sets the message handler (`h`), which is called when an error occurs. The specified data is passed to the handler. If no error handler is specified, a default handler that prints a message and exits will be used. Returns OK or ERROR. The message handler is defined by

```
typedef void
        (*DXLMessageHandler)(DXLConnection *conn, const char *msg,
                                   void *data);
```

16.7 Execution Control

These routines allow an application program to control the execution of an instance of Data Explorer (represented by `*conn`).

`DXLError DXLEndExecution(DXLConnection *conn)`
Terminates execution of a program running in Data Explorer. Returns OK or ERROR.

`DXLError DXLEndExecuteOnChange(DXLConnection *conn)`
Takes Data Explorer out of Execute on Change mode, but does not terminate the current execution.

`DXLError DXLExecuteOnChange(DXLConnection *conn)`
Puts Data Explorer into execute-on-change mode: the main macro reexecutes each time any of its inputs or referenced global variables changes value. Returns OK or ERROR.

`DXLError DXLEndExecuteOnChange(DXLConnection *conn);`
ends execute-on-change mode.

`DXLError DXLExecuteOnce(DXLConnection *conn)`
Initiates a single execution of the macro called main. Returns OK or ERROR.

`DXLError DXLGetExecutionStatus(DXLConnection *conn, int *state)`
Gets the execution status of Data Explorer. The parameter state is returned with a nonzero value if the executive is currently executing.

`DXLError DXLSequencerCtl(DXLConnection *conn, DXLSequencerCtlEnum action)`
Causes the specified action to occur. Valid arguments for `action` are:

SeqLoopOff
SeqLoopOn
SeqPalindromeOff
SeqPalindromeOn
SeqPause
SeqPlayBackward

SeqPlayForward
SeqStep
SeqStop

DXLError DXLSync(DXLConnection *conn)
Sends a message to the server and does not return until an acknowledgment is received. (This is a one-time synchronization. Compare with DXLSetSynchronization in 16.5, “Initialization and Exit” on page 169.

DXLError uiDXLSyncExecutive(DXLConnection *conn)
Sends a message through the user interface to the executive and does not return until the executive has acknowledged the message.

DXLError uiDXLSetRenderMode(DXLConnection *conn, char *title, int rmode)
lets you set the rendering mode of a window specified by **title**. The title of the window (accessible through the **Image Name** option in the **Options** pull-down menu of the Image window, the **title** parameter of the Image tool, or the **where** parameter of the Display module. **rmode** must be either 0 (for software) or 1 (for hardware).

DXLError uiDXLResetServer()
effectively does a “reset server”. That is, it flushes the Data Explorer software cache.

DXLError exDXLExecuteOnceNamed(DXLConnection *conn, char *name);

DXLError exDXLExecuteOnceNamedWithArgs(DXLConnection *conn, char *name, . ..);

DXLError exDXLExecuteOnceNamedWithArgsV(DXLConnection *conn, char *name, char **args);

DXLError exDXLExecuteOnChangeNamed(DXLConnection *conn, char *name);

DXLError exDXLExecuteOnChangeNamedWithArgs(DXLConnection *conn, char *name, ...);

DXLError exDXLExecuteOnChangeNamedWithArgsV(DXLConnection *conn, char *name , char **args);
These routines, used ONLY when the application is connected directly to the Data Explorer executive, allow the application to execute macros by name. The application can cause the named macro to be executed once or to be executed whenever a macro global input changes (such as when a value is sent to a DXLInput or DXLInputNamed module). The macro may be given a NULL-terminated list of arguments (the ...WithArgs form) or a NULL-terminated vector of arguments (the ...WithArgsV form).

16.8 Program Control

The following routines allow an application program to open visual programs and configuration files, set inputs and outputs of tools, and set up handlers for objects and values sent by the server to the calling program.

Loading programs and macros

DXLError exDXLBeginMacroDefinition(DXLConnection *conn, const char *mhdr)
Defines the beginning of a macro definition. The macro header ***mhdr** specifies the macro name, inputs, and outputs. For example:

mhdr = "macro SUM(arg1, arg2) → (sum)"

This routine should be followed by a series of DXLSend commands that send the macro definition, and finally by exDXLEndMacroDefinition (see following).

DXLError exDXLEndMacroDefinition(DXLConnection *conn)
Defines the end of a macro definition.

DXLError DXLLoadVisualProgram(DXLConnection *conn, const char *file)
Loads the visual program specified by the file name ***file**. The path to this file is relative to the startup directory of the server. Returns OK or ERROR.

If this routine is called when the application is communicating directly with the executive, an execution will occur after the visual program is loaded, because visual programs saved by the user interface include a call to the main macro.

DXLError DXLLoadMacroFile(DXLConnection *conn, const char *file);
Causes Data Explorer to load the macro contained in file 'file'.

DXLError DXLLoadMacroDirectory(DXLConnection *conn, const char *dir);
Causes Data Explorer to load all macros contained in directory 'dir'.

DXLError exDXLLoadScript(DXLConnection *conn, const char *file)
Loads the specified script (***file**) and executes it immediately. The path to this file is relative to the startup directory of the server. Returns OK or ERROR.

DXLError uiDXLLoadConfig(DXLConnection *conn, const char *file)
Opens the configuration file specified by its file name (***file**). The path to this file is relative to the startup directory of the user interface. Returns OK or ERROR.

Setting Variables

DXLink enables a program to set (and retrieve) Data Explorer values in a visual program or macro. It is important to understand the distinction between a Data Explorer object and a Data Explorer value. A Data Explorer object is the basic data structure of the Data Explorer data model (see Chapter 3, “Understanding the Data Model” on page 15 in *IBM Visualization Data Explorer User’s Guide*). A Data Explorer value is a character representation of a Data Explorer object (as would be used in the scripting language). The following are common examples of Data Explorer values:

```
      string:  "123"
     integer:  123
      scalar:  1.23
      vector:  [1 2 3]
 string list:  { "123", "456" }
integer list:  { 123, 456 }
 scalar list:  { 1.23, 4.56 }
 vector list:  { [1 2 3 ], [4 5 6] }
```

Not all Data Explorer objects can be represented by strings (e.g., fields and groups).

DXLError DXLSetValue(DXLConnection *conn, const char *varname,
const char *value)

Sets the global variable specified by ***varname** to the value given in ***value** (double quotation marks—for strings and string lists— must be escaped with a backslash (\), as in the example below.) Returns OK or ERROR.

This function is used primarily to set global variables in a macro and is the mechanism to set "inputs" to a module in a program.

Global variables are variables that have been assigned or referenced outside a macro (the global scope) or are defined by a reference in a nested scope. Variables assigned with a nested scope are considered local. For example, in the following macro, the variables **a** and **b** are local to the macro (their first occurrence is an assignment), while **c** is a global variable (its first occurrence is a reference).

```
macro foo( ) -> ( )
{
  a = "1";
  b = c;
. . . .
}
foo( );
```

See "Variables Used in Macros" on page 208 in *IBM Visualization Data Explorer User's Guide* for the rules of scoping variables.

You might use DXLSetValue to control the name of a data set imported with the Import tool, as in the following Data Explorer macro:

```
macro main( )
{
   object = Import(MyFileNameHere);
   ....
   Display(...)
}
```

The following C code invokes the main macro to import data from the file foobar.dx and render it:

```
. . .
DXLSetValue(conn, "MyFileNameHere", "\"foobar.dx\"");
DXLExecuteOnce(conn);
. . .
```

The Data Explorer user interface provides a convenient mechanism for placing global variables in a visual program. The DXLInput tool (see "DXLInput" on page 102 in *IBM Visualization Data Explorer User's Reference*) implements a global variable inside the macro main( ). By changing the label in the **Notation** field of the DXLInput tool's configuration dialog box, you can change the name of the global variable. This mechanism provides a clean interface between the visual program and the DXLink application. Named DXLInput tools are simply placed in the program and connected to the module input that needs to be controlled from the application. In the preceding example, a DXLInput tool named

MyFileNameHere would be connected to the first input of the Import tool. Then, the same piece of C code could be used to control the program.

```
DXLError DXLSetInteger(DXLConnection *conn, const char *varname, int value)
DXLError DXLSetScalar(DXLConnection *conn, const char *varname,
                      const double value)
DXLError DXLSetString(DXLConnection *conn, const char *varname,
                      const char *value)
```

Set the variable specified by ***varname** to **value** (or ***value**). These are convenience functions that use DXLSetValue. They return OK or ERROR.

Retrieving Values Sent From Data Explorer

The DXLOutput tool provides the means to retrieve values from Data Explorer asynchronously. This tool is used much like the macro Output tool in the Data Explorer user interface. It has two inputs: the first associates a label with an object or value (much like the name of a global variable); the second is the input object to be sent to the DXLink application. Currently, DXLOutput sends only those values that can be represented as strings.

When the module is executed, it communicates its input values to the DXLink application.

In order to retrieve the values in the application, a function must be defined and installed to accept them when they are available. A function is installed as follows:

```
DXLError DXLSetValueHandler(DXLConnection *c, const char *label,
                            DXLValueHandler h, const void *data);
```

where ***label** is the value of the first input to the corresponding DXLOutput for which the function is being installed. (In the user interface, the label corresponds to the **Notation** field in the configuration dialog box.) When the labeled value is received, the function, or handler **h**, is called as follows:

```
(*h) (c, label, value, data)
```

where **label** and **data** are the values that were passed to DXLSetValueHandler( ); **data** is a user-defined value, and **value** is the value being received for the corresponding label. The handler (like message handlers in general) is called when DXLHandlePendingMessages( ) is called and a corresponding message is pending.

The definition of the value-handler function is:

```
typedef void (*DXLValueHandler)(DXLConnection *conn, const char *label,
                                const char *value, void *data);
```

Note: The same handler can be installed for values with different labels. See also DXLRemoveValueHandler().

```
DXLError DXLRemoveValueHandler(DXLConnection *c, const char *label);
```

specifies that the value handler (**h**) for the value ***label** is to be removed (***label** is the name associated with the value that is to be passed to DXLink). Returns OK or ERROR.

16.9 Window Control

The following routines allow an application to control the behavior of user interface windows.

```
DXLError uiDXLOpenVPE(DXLConnection *conn);
```

```
DXLError uiDXLCloseVPE(DXLConnection *conn);
```

```
DXLError uiDXLOpenSequencer(DXLConnection *conn);
```

```
DXLError uiDXLCloseSequencer(DXLConnection *conn);
```

```
DXLError uiDXLOpenAllImages(DXLConnection *conn);
```

```
DXLError uiDXLCloseAllImages(DXLConnection *conn);
```

```
DXLError uiDXLOpenColorMapEditorByTitle(DXLConnection *conn, char *title);
```

```
DXLError uiDXLCloseAllColorMapEditors(DXLConnection *conn);
```

```
DXLError uiDXLCloseColorMapEditorByLabel(DXLConnection *conn, char *label);
```

```
DXLError uiDXLCloseColorMapEditorByTitle(DXLConnection *conn, char *title);
```

When the application is connected to the Data Explorer user interface, these routines allow control over DXUI windows. See “ManageImageWindow” on page 206 and and “ManageColormapEditor” on page 203 in *IBM Visualization Data Explorer User’s Reference* for a description of the difference between “title” and “label”.

Appendix A. Data Explorer Libraries

The most common use of the Data Explorer development libraries is to create your own modules, but there are several other ways they can be used.

A.1 libDXlite.a

To create your own data filters or stand alone data processing programs, as shown by the “DX Data Model” layer in Figure 1 on page 2, link your programs with the libDXlite.a library. This is the appropriate layer to use if your program wants to create, manipulate, import or export Data Explorer objects. None of the module routines nor any of the functions supplied by the Data Explorer Executive are included at this level.

If you are building a module to use with Data Explorer, you typically link with this library.

The routines contained in libDXlite.a are listed in Appendix B, “Data Explorer Data Model Library: DXlite Routines” on page 181.

A.2 libDXcallm.a

To create your own stand-alone programs which call DX modules, as shown by the “DX Modules” layer in Figure 1 on page 2, link your programs with the libDXcallm.a library. Because your program is the main routine and does not include any of the functions which are supplied by the Data Explorer Executive, you must do all object management, flow control (which modules to call when), any looping or conditional execution, and object deletion and reference count management. See “DXCallModule, DXModSet..., DXSetModule...” on page 203 for more information.

If you are building a module to use with Data Explorer, you would link with this library if you wanted to use DXCallModule to use some of the Data Explorer modules from within your own program. In addition, some higher level processing functions such as interpolation support, is available only in libDXcallm.a.

The routines contained in libDXcallm.a are listed in Appendix C, “Data Explorer Library Routines” on page 183.

A.3 libDXL.a

You can control the Data Explorer Executive from another program using the DXLink library libDXL.a. At this level, you can create visual programs using the DX VPE ahead of time and run them using only the DX Executive while providing your own user interface using any third party GUI builder or X library code. With “SuperviseWindow” on page 336 and “SuperviseState” on page 332, discussed in *IBM Visualization Data Explorer User’s Reference*, the functions you can control include the Image window interactions, and inserting an active image window into an application with an existing user interface. You get the object cache, the intelligent flow control, the object management and all the other functions which the executive provides.

You can also control the Data Explorer User Interface from another program using the libDXL.a DXLink library. In addition to the executive functions, you can use the Data Explorer Control Panels for user interaction, and the Image window with the direct interactions. The routines contained in libDXL.a are listed in Chapter 16, “DXLink Developer's Toolkit” on page 157. Figure 14 depicts the Data Explorer architecture.

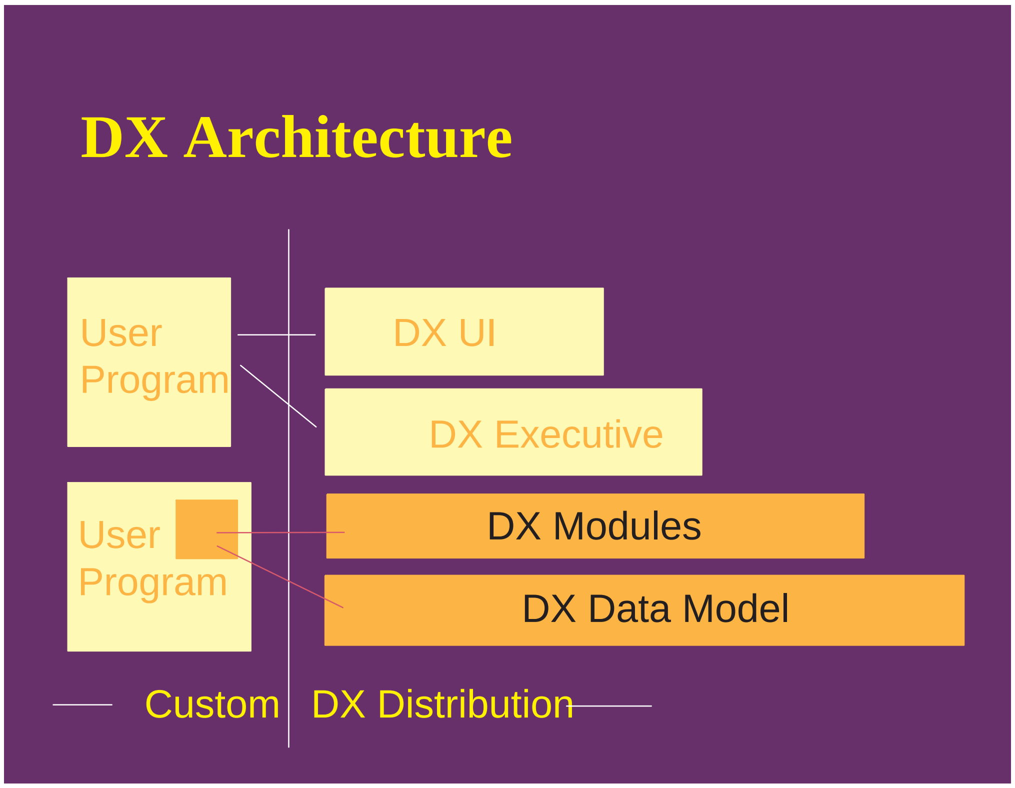


Figure 14. Data Explorer architecture

Appendix B. Data Explorer Data Model Library: DXlite Routines

All Data Explorer routines are included in the libraries libDXcallm.a and libDX.a. All routines are described in Appendix C, "Data Explorer Library Routines" on page 183. A subset of those routines are included in libDXlite.a, as shown in the list below. Outboard and stand-alone modules can use the Data Explorer data model by linking to this Library. You can create, query, and modify Data Explorer Objects. For example, you can write an importer that reads data in a particular file format and creates a Data Explorer Field Object as its output. (Outboard and stand-alone programs may also link to libDXcallm.a to access all of the routines described in Appendix C, "Data Explorer Library Routines" on page 183.)

DXAbortTaskGroup
DXAdd
DXAddArrayData
DXAddBackColor
DXAddBackColors
DXAddColor
DXAddColors
DXAddFaceNormal
DXAddFaceNormals
DXAddFrontColor
DXAddFrontColors
DXAddLine
DXAddLines
DXAddMessage
DXAddNormal
DXAddNormals
DXAddOpacities
DXAddOpacity
DXAddPoint
DXAddPoints
DXAddQuad
DXAddQuads
DXAddTask
DXAddTetrahedra
DXAddTetrahedron
DXAddTriangle
DXAddTriangles
DXAdjointTranspose
DXAllocate
DXAllocateArray
DXAllocateLocal
DXAllocateLocalOnly
DXAllocateLocalOnlyZero
DXAllocateLocalZero
DXAllocateZero
DXApply
DXArrayConvert
DXArrayConvertV

DXBeginLongMessage
DXBoundingBox

DXCategorySize
DXChangedComponent-Values
DXClipBox
DXClipPlane
DXColorNameToRGB
DXComponentOpt
DXComponentOptLoc
DXComponentReq
DXComponentReqLoc
DXConcatenate
DXCopy
DXCopyAttributes
DXCreateArrayHandle
DXCreateHash
DXCreateTaskGroup
DXCross

DXDebug
DXDelete
DXDeleteAttribute
DXDeleteComponent
DXDeleteHashElement
DXDestroyHash
DXDeterminant
DXDiv
DXDot

DXEmptyField
DXEnableDebug
DXEndField
DXEndLongMessage
DXEndObject
DXExecuteTaskGroup
DXExists

DXExportDX
DXExtract
DXExtractFloat
DXExtractInteger
DXExtractNthString
DXExtractParameter
DXExtractString

DXFree
DXFreeArrayDataLocal
DXFreeArrayHandle
DXFreeModuleId

DXGetArrayClass
DXGetArrayData
DXGetArrayDataLocal
DXGetArrayInfo
DXGetAttribute
DXGetBackgroundColor
DXGetCacheEntry
DXGetCacheEntryV
DXGetCameraMatrix
DXGetCameraMatrix-WithFuzz
DXGetCameraResolution
DXGetCameraRotation
DXGetClipBox
DXGetClippedInfo
DXGetComponentAttribute
DXGetComponentValue
DXGetConnections
DXGetConstantArrayData
DXGetEnumeratedAttribute
DXGetEnumerated-ComponentAttribute
DXGetEnumerated-ComponentValue
DXGetEnumeratedMember

DXGetError
DXGetErrorMessage
DXGetFloatAttribute
DXGetGroupClass
DXGetIntegerAttribute
DXGetItemSize
DXGetMember
DXGetMemberCount
DXGetMeshArrayInfo
DXGetMeshOffsets
DXGetModuleId
DXGetNextHashElement
DXGetObjectClass
DXGetObjectTag
DXGetOrthographic
DXGetPart
DXGetPartClass
DXGetPathArrayInfo
DXGetPathOffset
DXGetPerspective
DXGetPrivateData
DXGetProductArrayInfo
DXGetRegularArrayInfo
DXGetScreenInfo
DXGetSeriesMember
DXGetString
DXGetStringAttribute
DXGetTime
DXGetType
DXGetVaidCount
DXGetView
DXGetXformInfo

DXImportDX
DXInitNextHashElement
DXInsert

DXlite

DXInsertHashElement
DXInvert

DXLength
DXLn

DXMakeGridConnections
DXMakeGridConnectionsV
DXMakeGridPositions
DXMakeGridPositionsV
DXMakeStringList
DXMakeStringListV
DXMarkTime
DXMarkTimeLocal
DXMat
DXMax
DXMessage
DXMin
DXMul

DXNeg
DXNeighbors
DXNewAmbientLight
DXNewArray
DXNewArrayV
DXNewCamera
DXNewClipped
DXNewCompositeField
DXNewConstantArray
DXNewConstantArrayV
DXNewDistantLight
DXNewField
DXNewMeshArray
DXNewMultiGrid
DXNewPathArray
DXNewPrivate
DXNewProductArray
DXNewProductArrayV
DXNewRegularArray
DXNewScreen
DXNewSeries
DXNewString
DXNewXform
DXNormalize

DXPartition
DXPrint
DXPrintAlloc
DXPrintTimes
DXPrintV
DXProcessorId
DXProcessors
DXProcessParts
DXPt

DXQuad
DXQueryArrayCommon
DXQueryArrayCommonV
DXQueryArrayConvert
DXQueryArrayConvertV
DXQueryConstantArray
DXQueryDebug
DXQueryDistantLight
DXQueryGridConnections
DXQueryGridPositions
DXQueryHashElement
DXQueryParameter

DXReadyToRun
DXReAllocate
DXReference
DXRegisterInputHandler
DXRemove
DXRename
DXReplace
DXResetError
DXRotateX
DXRotateY
DXRotateZ

DXScale
DXSetAttribute
DXSetBackgroundColor
DXSetCacheEntry
DXSetCacheEntryV
DXSetClippedObjects
DXSetComponentAttribute
DXSetComponentValue
DXSetConnections
DXSetEnumeratedMember
DXSetError
DXSetFloatAttribute
DXSetGroupType
DXSetGroupTypeV
DXSetIntegerAttribute
DXSetMember
DXSetMeshOffsets
DXSetObjectTag
DXSetOrthographic
DXSetPart
DXSetPathOffset
DXSetPerspective
DXSetResolution
DXSetScreenObject
DXSetSeriesMember
DXSetStringAttribute
DXSetView
DXSetXformObject
DXSub
DXSwap

DXTetra
DXTraceTime
DXTranslate
DXTranspose
DXTri
DXTrim
DXTypeCheck
DXTypeCheckV
DXTypeSize

DXUnreference
DXUnsetGroupType

DXVec

DXWarning

Appendix C. Data Explorer Library Routines

The descriptions of routines begin after the list below, are sorted alphabetically by the name of the (first) routine, and contain the following information:

- General function
- Syntax
- Functional details
- Return value(s)
- Related routine(s) and information.

In entries that describe more than one routine, the second and any subsequent routines are not listed in the expected alphabetical order in the appendix. Their names, along with the page numbers of the entries in which they appear, are listed here.

A name ending in "(s)" signifies that the relevant entry describes both a "singular" and a "plural" version of a routine.

DXAbortTaskGroup

Function

Aborts a task Group without executing it.

Syntax

```
#include <dx/dx.h>

Error DXAbortTaskGroup()
```

Functional Details

If the creation of a task Group must be terminated before it can be completed, **DXAbortTaskGroup** should be called to release the resources allocated to that Group and its component tasks. After this call, the routine **DXExecuteTaskGroup** must not be called until a new task Group has been created (i.e., by **DXCreateTaskGroup** and **DXAddTask**).

Return Value

Returns **OK** or returns **ERROR** and sets an error code.

See Also

DXAddTask, **DXCreateTaskGroup**, **DXExecuteTaskGroup**

12.8, “Parallelism” on page 123.

DXAdd, DXCross, DXDiv, DXDot, DXLength, DXMax, DXMin, DXMul, DXNeg, DXNormalize, DXSub

Function

Perform standard vector mathematics.

Syntax

```
#include <dx/dx.h>

Vector DXNeg(Vector v)
Vector DXNormalize(Vector v)
double DXLength(Vector v)
Vector DXAdd(Vector v, Vector w)
Vector DXSub(Vector v, Vector w)
Vector DXMin(Vector v, Vector w)
Vector DXMax(Vector v, Vector w)
Vector DXMul(Vector v, double f)
Vector DXDiv(Vector v, double f)
float DXDot(Vector v, Vector w)
Vector DXCross(Vector v, Vector w)
```

.

Functional Details

DXNeg, DXNormalize, DXLength
Perform unary operations of negation, normalization, and length.

DXAdd, DXSub, DXMin, DXMax
Perform vector operations of addition, subtraction, min, and max. Min and max are performed on each component of a vector.

DXMul, DXDiv
Multiply or divide a vector by a float.

DXDot, DXCross
Form the dot product or cross-product of two vectors.

A **Point** and **Vector** are defined as follows:

```
typedef struct point {
    float x, y, z;
} Point, Vector;
```

Return Value

Each routine returns the result of its operation.

See Also

DXAdjointTranspose, **DXApply**, **DXConcatenate**, **DXDeterminant**, **DXInvert**, **DXTranspose**

“Basic Operations” on page 126.

DXAddArrayData

Function

Adds items to an Array.

Syntax

```
#include <dx/dx.h>

Array DXAddArrayData(Array a, int start, int n, Pointer data)
```

.

Functional Details

Adds **n** items to Array **a**, starting at the numbered item specified by **start**. The additions may replace or supplement items already defined. If **data** is not **NULL**, the routine copies the data into the Array; otherwise, it increases the number of items in the Array by **n**, but leaves them uninitialized.

To avoid having memory allocated every time, **DXAddArrayData** may allocate more than is needed for the requested number of items. If Array **a** is part of a Field, any extra space will be freed when **DXEndField** is called. If it is not part of a Field, **DXTrim** can be called to free the extra space, though this is not required.

To allocate space “up front” for an Array of known size, use **DXAddArrayData(0, n, NULL)**, which is the preferred means for preallocating space for the data.

DXAllocateArray(0, n, NULL) also preallocates space, but the number of items in the Array will be 0. If **DXAllocateArray** is called, **DXTrim** can be called to resize the Array to the space required for the actual number of items.

Allocating more space for an Array may result in its being copied to a new location in memory. If you call **DXGetArrayData** to obtain a pointer for an Array and then allocate more space for it, you must call **DXGetArrayData** again because the pointer may no longer be valid.

There are four ways to add data to irregular Arrays:

- Add items one at a time: **DXAddArrayData(a, i, 1,item)**.
- Add items in batches: **DXAddArrayData(a, i, n, items)**.
- Add multiple items all at one time: **DXAddArrayData(a, 0, n, items)**.
- Allocate the memory: call **DXAddArrayData(a, 0, n, NULL)**; get a pointer to global memory with **DXGetArrayData(a)**; and put the items directly into global memory "by hand."

Return Value

Returns **a** or returns **NULL** and sets an error code.

See Also

DXAllocateArray, **DXEndField**, **DXGetArrayData**, **DXTrim**,

"Irregular Arrays" on page 101.

DXAddFaceNormal, DXAddFaceNormals

Function

Adds connection-dependent normals to a Field.

Syntax

```
#include <dx/dx.h>

Field DXAddFaceNormal(Field f, int id, Vector v)
Field DXAddFaceNormals(Field f, int start, int n, Vector *v)
```

.

Functional Details

A Field may contain connection-dependent normals that can be used to flat-shade a polygonal object. If so, the normals component is expected to have the same size as the "connections" component, as indicated by its having a "dep" attribute of "connections." Both routines aid in constructing such a component.

DXAddFaceNormal
: Adds one normal (**v**) to **f** with the specified zero-based **id**. If **f** does not contain a "normals" component, one is added.

DXAddFaceNormals
: Adds **n** normals (***v**) to **f**. Identifiers begin with **start**. If **f** does not contain a "normals" component, one is added.

Note: Both routines are suitable for adding a small number of face normals and for rapid prototyping, but they are included here mainly for backward compatibility. For better performance, see "DXAddArrayData" on page 190 and "Irregular Arrays" on page 101.

Normals are specified as **`Vectors`** and defined as follows:

```
typedef struct point {
    float x, y, z;
} Point, Vector;
```

Note: These routines do not check the "dep" attribute of the "normals" component. Thus, if a "normals" component that is "dep" on "positions" already exists in **`f`**, the routine adds one or more normals to the normals component, leaving the attribute unchanged. The result may be a component with the wrong number of items for "dep."

Return Value

Returns **`f`** or returns **`NULL`** and sets an error code.

See Also

`DXAddArrayData`, **`DXAddBackColor`**, **`DXAddBackColors`**, **`DXAddColor`**, **`DXAddColors`**, **`DXAddFrontColor`**, **`DXAddFrontColors`**, **`DXAddNormal`**, **`DXAddNormals`**, **`DXAddOpacities`**, **`DXAddOpacity`**, **`DXAddPoint`**, **`DXAddPoints`**, **`DXAddArrayData`**

"Points and Dependent Data" on page 106.

DXAddLine, ...Triangle, ...Quad, ...Tetrahedron, ...Lines, ...Triangles, ...Quads, ...Tetrahedra

Function

Adds interpolation element(s) to a Field.

Syntax

```
#include <dx/dx.h>

Field DXAddLine(Field f, int id, Line l)
Field DXAddTriangle(Field f, int id, Triangle t)
Field DXAddQuad(Field f, int id, Quadrilateral q)
Field DXAddTetrahedron(Field f, int id, Tetrahedron t)

Field DXAddLines(Field f, int start, int n, Line *l)
Field DXAddTriangles(Field f, int start, int n, Triangle *t)
Field DXAddQuads(Field f, int start, int n, Quadrilateral *q)
Field DXAddTetrahedra(Field f, int start, int n, Tetrahedron *t)
```

.

Functional Details

The interpolation elements generated by these routines are stored in the "connections" component.

DXAddLine, DXAddTriangle, DXAddQuad, DXAddTetrahedron
Add a single line, triangle, quad, or tetrahedron to **f** with the specified zero-based **id**. If necessary, a routine creates the “connections” component.

DXAddLines, DXAddTriangles, DXAddQuads, DXAddTetrahedra
Add **n** lines, triangles, quads, or tetrahedra to **f**. Identifiers begin with **start**. If necessary, a routine creates the “connections” component.

Lines, Triangles, Quadrilaterals, and Tetrahedra. are defined as follows:

```
typedef struct line {
    PointId p, q;
} Line;

typedef struct triangle {
    PointId p, q, r;
} Triangle;

typedef struct quadrilateral {
    PointId p, q, r, s;
} Quadrilateral;

typedef struct tetrahedron {
    PointId p, q, r, s;
} Tetrahedron;
```

Note: It is an error to attempt adding one kind of interpolation element to a “connections” component that already contains a different kind.

Return Value

Returns **f** or returns **NULL** and sets an error code.

See Also

DXGetConnections, **DXSetConnections**

“Connections” on page 107.

DXAddMessage, DXMessageReturn, DXMessageGoto

Function

Concatenates a message with the current error message.

Syntax

```
#include <dx/dx.h>

Error DXAddMessage(char *message, ...)
#define DXMessageReturn(s) {DXAddMessage(s); return ERROR;}
#define DXMessageGoto(s) {DXAddMessage(s); goto error;}
```

.

Functional Details

This routine can provide more information about an error originally detected in a low-level routine. The contents of **message** may be a **printf** format string, in which case additional arguments required by the format string must be specified.

Return Value

Always returns **ERROR**.

See Also

DXSetError

12.1, "Error Handling and Messages" on page 114.

DXAddPoint, ...Color, ...FrontColor, ...BackColor, ...Opacity, ...Normal, DXAddPoints, ...Colors, ...FrontColors, ...BackColors, ...Opacities, ...Normals

Function

Add points or point-dependent data to a Field.

Syntax

```
#include <dx/dx.h>

Field DXAddPoint(Field f, int id, Point p)
Field DXAddColor(Field f, int id, RGBColor c)
Field DXAddFrontColor(Field f, int id, RGBColor c)
Field DXAddBackColor(Field f, int id, RGBColor c)
Field DXAddOpacity(Field f, int id, double o)
Field DXAddNormal(Field f, int id, Vector v)

Field DXAddPoints(Field f, int start, int n,  Point *p)
Field DXAddColors(Field f, int start, int n, RGBColor *c)
Field DXAddFrontColors(Field f, int start, int n, RGBColor *c)
Field DXAddBackColors(Field f, int start, int n, RGBColor *c)
Field DXAddOpacities(Field f, int start, int n, float *o)
Field DXAddNormals(Field f, int start, int n, Vector *v)
```

.

Functional Details

Associated with a Field may be a number of components that correspond one-to-one with the "positions" component, as indicated by their each having a "dep" attribute of "positions." These routines aid in constructing such components.

DXAddPoint, DXAddColor, DXAddFrontColor, DXAddBackColor, DXAddOpacity, DXAddNormal,

Add one point (position), color, front color, back color, opacity, or normal to **f** with the specified zero-based **id**. If necessary, the routine creates the appropriate component.

DXAddPoints, DXAddColors, DXAddFrontColors, DXAddBackColors, DXAddOpacities, DXAddNormals

Add **n** points, colors, front colors, back colors, opacities, or normals to **f** with zero-based identifiers beginning with **start**. If necessary, the routine creates the appropriate component.

Colors are specified as **RGBColors** and defined as follows:

```
typedef struct rgbcolor {
   float r, g, b;
} RGBColor;
```

Points and **Normals** are defined as follows:

```
typedef struct point {
   float x, y, z;
} Point, Vector;
```

Opacities are specified as floating-point values.

Notes:

1. These routines are suitable for adding a small number of points and for rapid prototyping. For better performance, see the description of **DXAddArrayData** and the discussion of direct-access routines in “Irregular Arrays” on page 101
2. These routines do not check the “dep” attribute of the component being added to; thus, these routines do not perform correctly if the component exists and has a dep “connections” attribute.

Return Value

Returns **f** or returns **NULL** and sets an error code.

See Also

DXAddArrayData, **DXAddFaceNormal**, **DXAddFaceNormals**, **DXGetArrayData**

“Points and Dependent Data” on page 106.

DXAddTask

Function

Adds a task to the current task Group.

Syntax

```
#include <dx/dx.h>

Error DXAddTask(Error(*proc)(Pointer), Pointer arg, int size, double work)

.
```

Functional Details

The task to be added is defined by the following parameters:

proc A pointer to a function that performs the task. It takes one argument of type **Pointer**.

arg A pointer to an argument block.

size The size of the argument block in bytes. If **size** is nonzero, the argument block is copied. Otherwise, the value of **arg** is passed to **proc** without copying, which is useful for passing integers or Objects.

work The estimated amount of time required by the task in arbitrary units.

Tasks are executed, in parallel if possible, by **DXExecuteTaskGroup**. Executing a task consists of calling **(*proc)(arg)**. To facilitate load balancing, tasks are executed in order of decreasing work estimates. Once **DXExecuteTaskGroup** has been called, additional tasks can be created only from other tasks in the current task Group, and they are run as soon as possible, without regard to **work** or the status of the original tasks.

Return Value

Returns **OK** or returns **ERROR** and sets an error code.

See Also

DXAbortTaskGroup, **DXCreateTaskGroup**, **DXExecuteTaskGroup**, **DXProcessorId**, **DXProcessors**

12.8, “Parallelism” on page 123.

DXAllocate, DXAllocateZero, DXAllocateLocal, DXAllocateLocalZero, DXAllocateLocalOnly, DXAllocateLocalOnlyZero

Function

Allocate global or local memory.

Syntax

```
#include <dx/dx.h>

Pointer DXAllocate(unsigned int n)
Pointer DXAllocateZero(unsigned int n)
Pointer DXAllocateLocal(unsigned int n)
Pointer DXAllocateLocalZero(unsigned int n)
Pointer DXAllocateLocalOnly(unsigned int n)
Pointer DXAllocateLocalOnlyZero(unsigned int n)
```

.

Functional Details

For all of these routines, **n** must be greater than zero.

DXAllocate
Allocates **n** bytes of memory in global memory.

DXAllocateZero
Allocates and clears **n** bytes of global memory.

DXAllocateLocal
On platforms with per-processor local memory (such as multiprocessor workstations), allocates **n** bytes of memory in local memory.

DXAllocateLocalZero
Allocates **n** bytes in local memory and zeros the allocated memory.

DXAllocateLocal, DXAllocateLocalZero
If **n** bytes of local memory are not available, allocate them from global memory.

DXAllocateLocalOnly, DXAllocateLocalOnlyZero
Allocate local memory only.

On platforms without per-processor local memory (such as all currently supported workstations), **DXAllocateLocal** is identical to **DXAllocate**.

Although local is usually faster than global access, local memory should be used only for Objects within a task or for parts of a module not in a task Group: Objects in local memory cannot be shared between processors. Objects that are the output of a module must be in global memory.

Note: Memory allocated by any routine listed here should be freed (with DXFree) when it is no longer needed.

Return Value

Returns a pointer to the allocated memory or returns **NULL** and sets an error code.

See Also

DXAbortTaskGroup, **DXAddTask**, **DXCreateTaskGroup**, **DXExecuteTaskGroup**, **DXFree**, **DXPrintAlloc**, **DXReAllocate**

12.3, “Memory Allocation” on page 116.

DXAllocateArray

Function

Allocates space for the data items of an Array.

Syntax

```
#include <dx/dx.h>

Array DXAllocateArray(Array a, int n)
```

.

Functional Details

This routine allocates to Array **a** the space for at least **n** items. It increases efficiency because it allocates space in advance, eliminating the need to allocate additional space at a later time. It is most useful when you can specify the maximum number of items that will be added. The routine changes only the amount of space allocated to Array **a**, not the *number of items* in it (only **DXAddArrayData** can do that).

Note: In the ideal case of knowing the *exact* number of items the Array will require, you can call **DXAddArrayData(a, o, n, NULL)**, which will allocate the right amount of space for the specified number of items (**n**). The allocation routine detailed here, on the other hand, is for creating Arrays when this information is *not* exact.

Return Value

Returns **a** or returns **NULL** and sets an error code.

See Also

DXAddArrayData, **DXTrim**

“Irregular Arrays” on page 101.

DXApplyTransform

Function

Creates a new Object by recursively applying a transform to a specified Object.

Syntax

```
#include <dx/dx.h>

Object DXApplyTransform(Object o, Matrix *m)
```

.

Functional Details

The routine descends Object **o**, applying transform **m** to the components of the Fields encountered. Any transforms present in the Object are accumulated to form a composite transform before being applied to subObjects. (The Object created can be deleted with DXDelete. See 2.4, “Memory Management” on page 13.)

DXApplyTransform returns a new Object (i.e., a copy of **o** with transforms applied). This routine can be applied directly to Array objects. It also removes the “box” component from transformed Fields.

Positional Components,
Are transformed by the composite matrix. (Positional components are “positions” and components with the “geometric” attribute “positional”.)

Vector Components,
Are transformed by the adjoint transpose of the composite matrix. (Vector components are “normals,” “binormals,” “tangents,” “gradient,” and components with the “geometric” attribute “vector.”)

Return Value

Returns **o** or returns **NULL** and sets an error code.

See Also

DXAdjointTranspose, **DXApply**, **DXConcatenate**, **DXDeterminant**, **DXInvert**, **DXMat**, **DXRotateX**, **DXRotateY**, **DXRotateZ**, **DXScale**, **DXTranslate**, **DXTranspose**

15.1, “Transformation” on page 150.

DXArrayConvert, DXArrayConvertV

Function

Creates a new Array of specified type, category, rank, and shape from an existing Array.

Syntax

```
#include <dx/dx.h>

Array DXArrayConvert(Array a, Type t, Category c, int rank, ...)
Array DXArrayConvertV(Array a, Type t, Category c, int rank, int *shape)
```

.

Functional Details

The routine copies Array **a** and converts it to type **t**, category **c**, rank **rank**, and a specified shape. (The new Array can be deleted with DXDelete. See 2.4, “Memory Management” on page 13.)

Conversion requires that the parameters of the newly created Array be compatible with those of the Array from which it was copied. For example, the new **rank** and **shape** are compatible with the rank and shape of **a** if they differ only by dimensions that have a shape of **1**. Thus an Array of **1** × **n** matrices can be converted to an Array of vectors. Table 2 and Table 3 summarize the convertibility of the different types and categories.

Table 2. Summary of Type Conversions

	Byte	**Unsigned Byte**	**Short**	**Unsigned Short**	**Int**	**Unsigned Int**	**Float**	**Double**
Byte	A	CNS	A	CNS	A	CNS	A	A
Unsigned Byte	CNS	A	A	A	A	A	A	A
Short	CNS	CNS	A	CNS	A	CNS	A	A
Unsigned Short	CNS	CNS	CNS	A	A	A	A	A
Int	CNS	CNS	CNS	CNS	A	CNS	A	A
Unsigned Int	CNS	CNS	CNS	CNS	CNS	A	A	A
Float	CNS	CNS	CNS	CNS	CNS	CNS	A	A
Double	CNS	CNS	CNS	CNS	CNS	CNS	CNS	A

Notes:

CNS = Conversion not supported
A = ANSI 'C' type-conversion semantics

Table 3 (Page 1 of 2). Summary of Category Conversions

	Real	**Complex**
Real	Conversion	Conversion

Table 3 (Page 2 of 2). Summary of Category Conversions

	Real	**Complex**
`Complex`	CNS	Conversion

Notes:

CNS = Conversion not supported
Real→Complex: a → a + 0i

Return Value

Returns a new Array or returns **`NULL`** and sets an error code.

See Also

`DXExtractFloat`, **`DXExtractInteger`**, **`DXExtractNthString`**, **`DXExtractParameter`**, **`DXExtractString`**, **`DXQueryArrayCommon`**, **`DXQueryArrayCommonV`**, **`DXQueryArrayConvert`**, **`DXQueryArrayConvertV`**, **`DXQueryParameter`**

11.8, "Extracting Module Parameters" on page 108.

DXBeginLongMessage, DXEndLongMessage

Function

Enables multiple **DXMessage** calls to create a single long message.

Syntax

```
#include <dx/dx.h>

void DXBeginLongMessage()
void DXEndLongMessage()
```

Functional Details

The **DXMessage** routine is suitable for relatively short, unformatted messages. For long, multiple-line messages, you may enclose a series of calls to **DXMessage** between **DXBeginLongMessage** and **DXEndLongMessage**. In this case, new lines are not automatically appended to each message, and it is your responsibility to indicate appropriate line breaks by including newline characters in the various **DXMessage** calls. For example, multiple calls to **DXMessage** may be printed on the same line, or one call to **DXMessage** may contain multiple lines.

Return Value

None.

See Also

DXMessage

12.1, “Error Handling and Messages” on page 114.

DXBoundingBox

Function

Computes the bounding box of an Object.

Syntax

```
#include <dx/dx.h>

Object DXBoundingBox(Object o, Point *box)
```

Functional Details

This routine adds to Object **o**, and any of its descendants that are Fields, a “box” component consisting of an Array of 2^d points that are the corners of a bounding box (where *d* is the dimensionality of the data). For data of dimensionality three or less, the routine returns—in the Array pointed to by **box**—the eight corner points; for dimensionalities of one or two, the extra dimensions are treated as zero in the box returned by the routine.

The bounding box returned on **o** is determined by combining the bounding boxes of all Fields contained in **o**.

Transformations are considered in computing the bounding box, but clipping Objects are not. The bounding box is not guaranteed to be the tightest possible.

A **Point** is defined as follows:

```
typedef struct point {
   float x, y, z;
} Point, Vector;
```

Return Value

Returns **o** or returns **NULL** and may or may not set an error code, depending on the input. For example, it does not set an error code if a bounding box cannot be defined for the given input.

See Also

DXChangedComponentStructure, **DXChangedComponentValues**, **DXEmptyField**, **DXEndField**, **DXEndObject**, **DXNeighbors**, **DXStatistics** **DXValidPositionsBoundingBox**

"Standard Components" on page 107.

DXCallModule, DXModSet..., DXSetModule...

Function

Enable a routine to call a Data Explorer module.

Syntax

```
#include <dx/dx.h>
 Error DXCallModule(char *modname, int num_inputs, ModuleInput *listin,
                                  int num_outputs, ModuleOutput *listout);

Object DXModSetFloatInput(ModuleInput *in, char *name, int n);
Object DXModSetIntegerInput(ModuleInput *in, char *name, int n);
Object DXModSetStringInput(ModuleInput *in, char *name, char *s);
  void DXModSetObjectInput(ModuleInput *in, char *name, Object obj);
  void DXModSetObjectOutput(ModuleOutput *out, char *name, Object *obj);

       DXSetModuleInput(ModuleInput in, char *name, Object *obj);
       DXSetModuleOutput(ModuleOutput out, char *name, Object *obj);
```

Functional Details

The five **DXModSet...** routines are auxiliary to **DXCallModule**, which makes the actual call to a specified module.

Notes:

1. If you use **DXCallModule** in a stand-alone program or outboard module, you must call **DXInitModule** before making any calls to **DXCallModule**
2. The two **DXSetModule...** routines are provided solely for backward compatibility. Their use is not recommended.
3. The Get and Set modules cannot be called by DXCallModule.
4. Objects passed as inputs to DXCallModule will be deleted when that module is finished. To use the Object after DXCallModule requires a call to DXReference first, and the responsibility for deleting the Object when you are finished with it is yours.

modname specifies the name of the module being called.

num_inputs specifies the number of inputs in **listin**.

listin is an Array of ModuleInput structures specifying the module inputs.

num_outputs specifies the number of outputs in **listout**.

listout is an Array of ModuleOutput structures specifying the module outputs.

The four **DXModSet...Input** routines set the contents of **ModuleInput** structures. **DXModSetObjectOutput** sets the destination for module outputs.

- If **name** is specified as **NULL**, the inputs and outputs set by these routines are considered positional. For example, the first values passed using one of the **DXModSet...Input** routines will be assigned to the first parameter of the module, the second value to the second parameter of that module, and so on.
- If **name** is a valid string, then it specifies a parameter of the module specified by **modname**, and the passed value is assigned to that parameter.

- Positional parameters cannot follow named ones.

A **ModuleInput** is defined as follows:

```
typedef struct {
   char *name;
   Object value;
} ModuleInput;
```

A **ModuleOutput** is defined as follows:

```
typedef struct {
   char *name;
   Object *value;
} ModuleOutput;
```

Return Value

Returns **NULL** or returns **ERROR** and sets an error code.

See Also

DXInitModules,
DXGetErrorExit,
DXSetErrorExit,
DXCheckRIH

12.10, "Module Access" on page 127.

`Readme` file in `/usr/lpp/dx/samples/callmodule/Readme`.

DXChangedComponentValues, DXChangedComponentStructure

Function

Delete components of a Field.

Syntax

```
#include <dx/dx.h>

Field DXChangedComponentValues(Field f, char *component)
Field DXChangedComponentStructure(Field f, char *component)
```

Functional Details

DXChangedComponentValues deletes all the components of **f** that have a "der" attribute naming the specified component **component**. This routine is typically used when the values of the items of an Array change (e.g., the values of the "data" component) but not the number of items.

DXChangedComponentStructure deletes all the components of **f** that have a "dep," "der," or "ref" attribute naming the specified component **component**. This routine is typically used when the number of items in an Array (e.g., the number of items in the "positions" component) has been changed.

Both of these routines recursively apply **DXChangedComponentStructure** to the components they delete. They ensure that Fields remain internally consistent when they are altered.

By deleting components derived from a changed component, a call to **DXChangedComponentValues** ensures that the derived component will be recalculated when necessary and will remain up-to-date. For example, the "data statistics" component is derived from the "data" component. If the "data" component is changed, the current contents of data statistics become invalid. A call to **DXChangedComponentValues(field, "data")** will delete data statistics, and the values will be recomputed on the next call to **DXStatistics**.

Similarly, **DXChangedComponentStructure** ensures that components that depend on, refer to, or are derived from another component will be as up-to-date as possible. For example, the "connections" component refers to the "positions" component. If the structure of the "positions" component is changed, perhaps by deleting a position, the references in the "connections" component that are indices into the "positions" component cease to apply. Rather than leave an invalid "connections" component in the Field, it is better to remove it by calling **DXChangedComponentStructure**.

Note: Most components depend on, refer to, or are derived from others. These routines may cause important information to be discarded. It is often better to correct the component that has a "dep," "der," or "ref" attribute than to delete it. In the example just described, you can avoid deleting the "connections" component when a point is deleted from the "positions" component by deleting all connections elements that refer to the discarded position and remapping the remaining references to reflect the moved points in the "positions" component.

Return Value

Returns **f** or returns **NULL** and sets an error code.

See Also

DXEndField, DXGetComponentValue, DXSetComponentValue

"Standard Components" on page 107.

DXCheckRIH

Function

Checks registered input handlers.

Syntax

```
#include <dx/dx.h>

int DXCheckRIH(int block);
```

Functional Details

This routine must be called periodically in a stand-alone program if the **DXRegisterInputHandler** routine is used to define an input handler. (The executive provides this function automatically for built-in Data Explorer routines.)

DXCheckRIH determines whether any input handlers need to be called and, if so, calls them before returning. The valid arguments for **block** are:

0 = Check whether any events need handling. If not, return. Otherwise, handle the event(s) and return.
1 = Do not return until an event that requires handling occurs.

If the Display module is part of your stand-alone program, this routine must be called, since Display uses an input handler to deal with external events (e.g., repainting window contents after they have been obscured).

If the SuperviseState module ("SuperviseState" on page 332 in *IBM Visualization Data Explorer User's Reference*) is being used in conjunction with Display to implement direct interaction, **DXCheckRIH** must not be called between SuperviseState and Display. This will ensure that SuperviseState passes up-to-date state information into Display.

Return Value

Returns 1 if an event required handling; otherwise return 0.

See Also

DXRegisterInputHandler

DXClipBox

Function

Creates a clipping Object defined by a clipping box.

Syntax

```
#include <dx/dx.h>

Object DXClipBox(Object o, Point p1, Point p2)
```

Functional Details

Creates a new Object that defines a clipping transformation to be performed on Object **o** at render time. The Object will be clipped by the box, whose diagonal is defined by points **p1** and **p2**.

A **Point** is defined as follows:

```
typedef struct point {
   float x, y, z;
} Point, Vector;
```

Return Value

Returns an Object describing to the renderer how to clip at render time or returns **`NULL`** and sets an error code.

See Also

`DXClipPlane`

14.2, “Clipping” on page 146.

DXClipPlane

Function

Creates a clipping Object defined by a clipping plane.

Syntax

```
#include <dx/dx.h>

Object DXClipPlane(Object o, Point p, Vector n)
```

Functional Details

Creates a new Object that defines a clipping transformation to be performed on Object **o** at render time. The Object will be clipped by the plane that contains **p** and is perpendicular to vector **n**. The Object on the side of the plane pointed to by **n** is retained.

A **Point** or **Vector** is defined as follows:

```
typedef struct point {
   float x, y, z;
} Point, Vector;
```

Return Value

Returns an Object describing to the renderer how to clip at render time or returns **NULL** and sets an error code.

See Also

DXClipBox

14.2, “Clipping” on page 146.

DXColorNameToRGB

Function

Gets the RGB values for a specified color-name string.

Syntax

```
#include <dx/dx.h>

Error DXColorNameToRGB(char *colorname, RGBColor *rgbvalue)
```

Functional Details

The RGB values of the color string **colorname** are obtained from a color lookup table and returned in ***rgbvalue**. For example, the RGB values for the color “white” are 1.0, 1.0, 1.0. The color lookup table could come from a number of places:

- The routine first checks the environment variable DXCOLORS. If that is set, the routine uses the value of the variable and tries to open it as a file.

- If that does not succeed, the routine then checks the environment variables DXEXECROOT and DXROOT for a directory path. It will append `/lib/colors.txt` to this path and try to open the file. If that fails, it tries to open `/usr/lpp/dx/lib/colors.txt`.
- If none of these files can be opened, the routine uses an internal static table. This table is a subset of colors in the file `/usr/lpp/dx/lib/colors.txt`.

Return Value

Returns **OK** or returns **ERROR** and sets an error code.

See Also

"Colors" on page 125.

DXCompareModuleID

Function

Determines whether two module identifiers are the same.

Syntax

```
#include <dx/dx.h>

Error DXCompareModuleId(Pointer id1, id2);
```

Return Value

Returns OK, if the two identifiers are the same, or returns ERROR. This routine does not set an error code.

See Also

DXCopyModuleID, DXGetModuleID, DXFreeModuleId

12.5, "Cache" on page 121,
12.11, "Asynchronous Services" on page 129.

DXComponentReq, DXComponentOpt, DXComponentReqLoc, DXComponentOptLoc

Function

Access or typecheck a component in a Field.

Syntax

```
#include <dx/dx.h>

Error DXComponentReq(Array a, Pointer *data, int *n, int nreq, Type t, int dim)
Error DXComponentOpt(Array a, Pointer *data, int *n, int nreq, Type t, int dim)
Error DXComponentReqLoc(Array a, Pointer *data, int *n, int nreq, Type t, int dim)
Error DXComponentOptLoc(Array a, Pointer *data, int *n, int nreq, Type t, int dim)
```

Functional Details

Note: These routines will become obsolete in a future version of Data Explorer. In their place, it is recommended that you now use **`DXGetComponentValue`** and **`DXGetArrayInfo`**.

The four routines have identical calling sequences, but differ as follows.

First, **`DXComponentOpt`** and **`DXComponentReq`** return pointers to the global copy of the Array data, while **`DXComponentOptLoc`** and **`DXComponentReqLoc`** return pointers to a local copy of the Array data, and should be matched by a **`DXFreeArrayDataLocal`** call.

Second, **`DXComponentReq`** and **`DXComponentReqLoc`** consider it an error if the component is missing (**`a`** is **`NULL`**), while **`DXComponentOpt`** and **`DXComponentOptLoc`** consider the component optional and do not consider a **`NULL a`** to be an error.

If **`data`** is not **`NULL`**, a pointer to a global or local copy of the data is returned in **`*data`**. If **`n`** is not **`NULL`**, the number of items in the Array is returned in **`*n`**. If **`n`** is **`NULL`**, the number of Array items must be **`nreq`**. The type of the Array must be **`t`**. If **`dim`** is 0, the Array must have rank 0 (scalar). If **`dim`** is nonzero, the Array must have rank 1 and shape equal to **`dim`**.

The type is one of the following:

`TYPE_BYTE`	**`TYPE_HYPER`**	**`TYPE_SHORT`**
`TYPE_UBYTE`	**`TYPE_INT`**	**`TYPE_USHORT`**
`TYPE_DOUBLE`	**`TYPE_UINT`**	**`TYPE_STRING`**
`TYPE_FLOAT`		

Note: These functions will expand compact data Arrays and use extra memory.

Return Value

Returns **`OK`** or returns **`ERROR`** and sets an error code.

See Also

`DXGetArrayData, DXGetArrayInfo, DXGetComponentValue`

11.1, "Field Class" on page 97.

DXConcatenate, DXInvert, DXTranspose, DXAdjointTranspose, DXDeterminant, DXApply

Function

Apply standard matrix operations.

Syntax

```
#include <dx/dx.h>

Matrix DXConcatenate(Matrix s, Matrix t)
Matrix DXInvert(Matrix t)
Matrix DXTranspose(Matrix t)
Matrix DXAdjointTranspose(Matrix t)
```

```
float DXDeterminant(Matrix t)
Vector DXApply(Vector v, Matrix t)
```

Functional Details

DXConcatenate returns a Matrix that is equivalent to transformation matrix **s** followed by transformation matrix **t**.

DXInvert, DXTranspose, DXAdjointTranspose, and **DXDeterminant** compute the matrix inverse, transpose, and adjoint transpose, and determinant, respectively.

DXApply applies matrix transformation **t** to **v**.

A **Matrix** is defined as follows:

```
typedef struct matrix {
        /* xA + b * /
        float A[3][3]
        float b[3];
} Matrix
```

A **Point** and **Vector** is defined as follows:

```
typedef struct point {
    float x, y, z;
} Point, Vector;
```

Return Value

Always returns the result of the operation.

See Also

DXApplyTransform, DXRotateY, DXRotateZ, DXScale, DXTranslate

“Basic Operations” on page 126.

DXCopy

Function

Performs various copying operations.

Syntax

```
#include <dx/dx.h>

Object DXCopy(Object o, enum copy copy)
```

Functional Details

The DXCopy operations differ in the depth to which they copy the structure of an Object **o**. Depth is specified by the **copy** parameter, which may be one of the following:

- **COPY_STRUCTURE**:

- For Groups, copies the Group header and recursively copies all Group members.
- For Fields, copies the Field header but *does not* copy the components (which are generally Arrays); instead it puts references to the components of the given Object into the resulting Field.
- For Arrays, passes back a pointer to the data and makes no copy.

- **COPY_HEADER**: Copies only the header of the immediate Object but *does not* copy attributes, members, components, and so on; instead it puts references to them into the new Object.

  The Object created can be deleted with DXDelete. See 2.4, "Memory Management" on page 13.

- **COPY_ATTRIBUTES**: Creates a new Object of the same type as the old, and copies all attributes and type information, but *does not* put references (to members, components, and so on) in the new Object.

Notes:

1. Because of the data-flow execution model used by Data Explorer, it is critical that no module actually alter its inputs. Instead, **DXCopy** is generally used to create a modifiable copy of the input that is then altered and produced as output. This is most often done by using the **COPY_STRUCTURE** form. This produces a copy of the structure of the input, but uses references to the Arrays of the input, rather than actual copies. Once a structure has been copied in this manner, the Arrays (which are most often found as components of Fields) may be replaced in the copied Field by new results. The result will be a copy of the input that shares all *unchanged* data with the original, thereby saving memory space.
2. **DXCopy** applied to an Object of type Array performs no copy and simply returns its input. This presents the problem that in order to clean up, copied Objects of other types may be deleted without deleting the original, while the result of applying **DXCopy** to Arrays *may not* be deleted without deleting the original.

Return Value

Returns the copy or returns **NULL** and sets an error code.

See Also

DXCopyAttributes, DXNewField, DXNewGroup

"Object Routines" on page 119.

DXCopyAttributes

Function

Copies attributes from one Object to another.

Syntax

```
#include <dx/dx.h>

Object DXCopyAttributes(Object dst, Object src)
```

Functional Details

Gets the value of each attribute, in turn, from the source Object **src** and sets it in an identically named attribute in the destination Object **dst**. Attributes already present in **dst** that are also present in **src** have their values updated to those in **src**. The values of attributes present in **dst** but not present in **src** remain unchanged.

Return Value

Returns **dst** or returns **NULL** and sets an error code unless **dst** was also specified as **NULL**.

See Also

DXGetAttribute, DXGetEnumeratedAttribute, DXSetAttribute

“Object Routines” on page 119.

DXCopyModuleID

Function

Returns a pointer to a copy of a specified module identifier.

Syntax

```
#include <dx/dx.h>

Pointer DXCopyModule(Pointer id);
```

Functional Details

When it is no longer needed, the copy should be deleted with DXFreeModuleID.

Return Value

Returns the pointer or returns NULL (and sets an error code if an error occurs).

See Also

DXCompareModuleID DXGetModuleID, DXFreeModuleId

12.5, “Cache” on page 121,
12.11, “Asynchronous Services” on page 129

DXCreateArrayHandle

Function

Creates a “handle” to allow convenient access to the items in any Array class.

Syntax

```
#include <dx/dx.h>

ArrayHandle DXCreateArrayHandle(Array array)
```

Functional Details

There are three different approaches for writing functions that support all of the defined Array classes:

1. Use **DXGetArrayClass** to determine the specific Array class, and then use the class-specific functions (e.g., **DXGetConstantArrayData**). The advantage of this approach is it uses the most efficient method to access the data stored in the Array. The disadvantage is that you need to write different code for each Array class.
2. Use **DXGetArrayData** on all Arrays. The advantage here is the simplicity; the disadvantage is that **DXGetArrayData** expands compact data, greatly increasing memory use.
3. Use the Array handling routines. The advantage is that they work on Arrays of any class without expanding the compact data; the disadvantage, that they are slightly less efficient for some Array classes.

   The ArrayHandle created should be deleted with **DXFreeArrayHandle** when the user no longer needs it. See 2.4, “Memory Management” on page 13.

Return Value

Returns an Array Handle or returns **NULL** and sets an error code.

See Also

DXFreeArrayHandle, DXGetArrayEntries, DXGetArrayEntry, DXIterateArray

“Array Handling” on page 102.

DXCreateHash

Function

Creates a hash table for storing elements.

Syntax

```
#include <dx/dx.h>

HashTable DXCreateHash(int elementSize, PseudoKey (*hashFunc)(),
                       int (*cmpFunc)())
```

Functional Details

An element to be stored consists of a key plus whatever data is to be associated with that key. Its size, in bytes, is specified by **`elementSize`**. in bytes. The parameter **`hashFunc`** points to the optional hash function callback. Given an element, **`hashFunc`** should return a uniformly distributed long integer pseudokey. If **`hashFunc`** is not provided, then the first long integer word of each key is assumed to be the pseudokey. **`cmpFunc`** is the optional compare function callback. Given a search key and an element, **`cmpFunc`** should return 0 if the key matches the element. If no compare function is provided, any element matching the pseudokey is assumed to match the search key.

Optionally provided by the calling application:

```
PseudoKey hashFunc (Key key);
```

Called on insertion and query to convert the arbitrary-size search key into the long integer pseudokey used to store the hash table-element.

```
int cmpFunc (Key searchKey, Element element);
```

Called on insertion and query when an element from the table matches the pseudokey. Returns 0 if the search key matches the key contained in the element found in the table.

See 13.5, “Hashing” on page 139 for additional details on **`hashFunc`** and **`cmpFunc`**. The HashTable created should be deleted with DXDestroy when it is no longer needed. See 2.4, “Memory Management” on page 13.

`PseudoKey` is defined as:

```
typedef long PseudoKey;

typedef Pointer Key;
```

Return Value

Returns the hash table or returns **`NULL`** and sets an error code.

See Also

`DXDeleteHashElement, DXDestroyHash, DXGetNextHashElement, DXInitGetNextHashElement, DXInsertHashElement, DXQueryHashElement`

13.5, “Hashing” on page 139.

DXCreateInvalidComponentHandle

Function

Creates an invalid-component handle.

Syntax

```
#include <dx/dx.h>

InvalidComponentHandle DXCreateInvalidComponentHandle(Object object,
                                                      Array array, char *name)
```

Functional Details

The invalid-component handle is necessary in order to use the other invalid-component routines.

object specifies:

- a Field for which an invalid-component Array is to be created; *or*
- one of the following kinds of array: “positions,” “connections,” “face,” or “polylines.”

array allows an initial “invalid positions” or “invalid connections” component to be passed in to initialize the handle.

Note: If **object** is a Field, it is not necessary to specify **array**, since the initial invalid component will be found in the Field. However, if **object** is a Field and **array** is not **NULL**, then **array** supersedes the invalid Array in the Field.

name Specifies the component referred to by the invalid component: “positions,” “connections.,” “faces,” or “polylines.”

When you have finished using it, delete the invalid-component handle with **DXFreeInvalidComponentHandle**.

Return Value

Returns the invalid-component handle or returns **NULL** and sets an error code.

See Also

DXFreeInvalidComponentHandle, DXGetInvalidCount, DXGetValidCount, DXInvertValidity, DXIsElementInvalid, DXIsElementValid, DXSaveInvalidComponent, DXSetAllInvalid, DXSetAllValid, DXSetElementInvalid, DXSetElementValid

13.3, “Invalid Data” on page 133.

DXCreateTaskGroup

Function

Starts a new task Group to utilize parallelism.

Syntax

```
#include <dx/dx.h>

Error DXCreateTaskGroup()
```

Functional Details

All tasks subsequently created with **`DXAddTask`** until the matching **`DXExecuteTaskGroup`** is called will be members of this task Group. Each task added to this group will, when **`DXExecuteTaskGroup`** is called, be sorted by its work values and executed in parallel if possible.

If an error occurs before ExecuteTaskGroup is called, AbortTaskGroup should be called to free memory associated with the task group. See 2.4, “Memory Management” on page 13.

Return Value

Returns **`OK`** or returns **`ERROR`** and sets an error code.

See Also

`DXAbortTaskGroup, DXAddTask, DXExecuteTaskGroup, DXProcessorId, DXProcessors`

12.8, “Parallelism” on page 123.

DXCull

Function

Removes invalid positions, connections, faces, or polylines (and their dependent information) from the Fields of an Object.

Syntax

```
#include <dx/dx.h>

Object DXCull(Object object)
```

Functional Details

Validity is determined from the contents of the “invalid” components: positions, connections, faces, or polylines. In any components that are dependent on these, elements corresponding to removed positions, connections, faces, or polylines are themselves removed. In any components that reference positions, connections, faces, or polylines, indices are renumbered, with the value -1 inserted for indices that reference removed elements. The “invalid” components are removed.

In general, **`DXInvalidateConnections`** and **`DXInvalidateUnreferencedPositions`** should be called before **`DXCull`** is called. This ensures that all connection, face, or polyline elements that reference invalid positions will be removed, along with all positions no longer referenced by any connections, faces, or polylines.

Return Value

Returns the updated Object or returns **`NULL`** and sets an error code.

See Also

`DXInvalidateConnections, DXInvalidateUnreferencedPositions`

13.3, “Invalid Data” on page 133.

DXDebug, DXEnableDebug, DXQueryDebug

Function

Perform operations on global debug keys.

Syntax

```
#include <dx/dx.h>

void DXDebug(char *classes, char *message, ...)
void DXEnableDebug(char *classes, int enable)
int DXQueryDebug(char *classes)
```

Functional Details

DXDebug compares the array of 1-character keys in **classes** to the set of keys that have been enabled with **DXEnableDebug**. If it finds a match, **DXDebug** calls **DXMessage** with **message** and any parameters that follow **message**.

DXEnableDebug enables or disables (**enable** = 1 or 0, respectively) the global key corresponding to each key in the array of 1-character **keys**. Usually, this routine is not called directly but is accessed at run time by calling the **Trace** module (see *IBM Visualization Data Explorer User's Reference*).

DXQueryDebug compares the array of 1-character **keys** to the set of keys that have been enabled with **DXEnableDebug**. It returns 1 if any key matches; otherwise, it returns 0.

Note: The upper-case letters A–Z and the numbers 0–9 are reserved for system use. Module writers may use the lowercase letters a–z.

Example: If the module code for **MyModule()** contained the following lines:

```
DXDebug("aqr", "the value of the index is %d",i);
DXDebug("ar", "the last value was %d", last);
DXDebug("asq", "entering for loop");
DXDebug("a", "function foo() returned an error ");
```

Then after executing the following modules:

```
Trace("q",1);
MyModule();
```

Messages 1 and 3 would be printed.

If the following modules are then executed:

```
Trace("r",1);
MyModule();
```

Messages 1, 2, and 3 would be printed, since both "q" and "r" are now enabled.

If the following modules were then executed:

```
Trace("qr",0);
MyModule();
```

None of the messages would be printed, as “a,” “q,” “s,” and “r” are now all disabled.

Messages are printed using the **DXMessage** function (see “DXMessage” on page 296).

Return Value

DXDebug and **DXEnableDebug** have no return value. **DXQueryDebug** returns 0 or 1.

See Also

DXMessage

12.1, “Error Handling and Messages” on page 114.

DXDelete

Function

Deletes a reference to an Object.

Syntax

```
#include <dx/dx.h>

Error DXDelete(Object o)
```

Functional Details

A call to this routine indicates that Object **o** is no longer needed. A reference count is maintained inside each Object; the memory associated with each Object **o** is not actually released until the last user of Object **o** has called **DXDelete**. If **o** is **NULL**, then **DXDelete** immediately returns **OK**.

When **DXDelete** is called on Objects that contain other objects (e.g., Groups or Fields), it decrements the reference count on all Objects in the hierarchy. A module that creates an Object is responsible for deleting the Object unless it is either made part of another Object (e.g, using **DXSetComponentValue**) or it is returned as the module output.

Return Value

Returns **OK** or returns **ERROR** and sets an error code.

See Also

DXCopy, DXReference

2.4, “Memory Management” on page 13, “Object Routines” on page 119.

DXDeleteComponent

Function

Deletes a named component from a Field.

Syntax

```
#include <dx/dx.h>

Field DXDeleteComponent(Field f, char *component)
```

Functional Details

Deletes **component** from a Field **f**. Any attributes associated with the named component are also deleted from the Field.

Deleting a component from a Field may alter the structure of the Field in significant ways. For example, if the “positions” component is removed, any other components that contain references to positions become invalid. For this reason, **DXChangedComponentStructure** may be used to ensure that the remaining structure of the Field is consistent.

Return Value

Returns **f** or returns **NULL** but does not set an error code if the component does not exist.

See Also

DXChangedComponentStructure, DXGetEnumeratedComponentValue, DXNewField, DXSetComponentValue

11.1, “Field Class” on page 97.

DXDeleteHashElement

Function

Removes from a hash table any element that matches a search key.

Syntax

```
#include <dx/dx.h>

Error DXDeleteHashElement(HashTable hashtable, Key searchKey)
```

Functional Details

If a hash function was provided at the time the hash table was created, then that function will be used to derive a pseudokey from **searchKey**. If a hash function was not provided, then the first long integer word of **searchKey** is assumed to be the pseudokey.

If more than one element is stored under that pseudokey (possibly only if a compare function was provided at the time the hash table was created), then that compare function will be used to delete only that element that matches **searchKey**.

Key is defined as:

```
typedef Pointer Key;
```

Return Value

Always returns **OK**.

See Also

DXCreateHash, DXInsertHashElement

13.5, "Hashing" on page 139.

DXDestroyHash

Function

Deletes a hash table.

Syntax

```
#include <dx/dx.h>

Error DXDestoryHash(HashTable hashTable)
```

Functional Details

The routine deletes the table and frees all memory associated with it.

Return Value

Returns **OK** or returns **ERROR** and sets an error code.

See Also

DXCreateHash

13.5, "Hashing" on page 139.

DXDisplayX, DXDisplayX8, DXDisplayX12, DXDisplayX24

Function

Displays an image in an X window.

Syntax

```
#include <dx/dx.h>

Object DXDisplayX(Object i, char *xdisplay, char *title)
Object DXDisplayX8(Object i, char *xdisplay, char *title)
Object DXDisplayX12(Object i, char *xdisplay, char *title)
Object DXDisplayX24(Object i, char *xdisplay, char *title)
```

Functional Details

Displays image **i** in an X window on the display specified by **xdisplay**, with the title specified by **title**. **xdisplay** is used as the X display string when opening the window. The window associated with **xdisplay** is maintained for subsequent calls to **DXDisplayX** until the user closes it, after which a new window is created.

These routines can utilize 8-bit pseudo color X visuals, 12-bit Direct Color and True Color visuals, and 24-bit Direct Color or True Color visuals.

DXDisplayX tries to create a window with the default visual, and if it is not of an appropriate type, tries to create an 8-bit, then 12-bit, then 24-bit visual. The other routines try to create the appropriate depth window first (for example, **DXDisplayX8** tries to create an 8-bit window, then tries the default window depth.

Note: **title** cannot begin with a number or with two # characters.

Return Value

Returns **i** or returns **NULL** and sets an error code.

See Also

DXDisplayFB, DXMakeImage

15.9, "Image Fields" on page 156.

DXEmptyField

Function

Determines whether a Field contains information.

Syntax

```
#include <dx/dx.h>

int DXEmptyField(Field f)
```

Functional Details

A Field is considered to be empty under any of the following conditions:

- It has no components.
- It has no “positions” component.
- Its “positions” component does not contain any elements.

Modules may use this call to avoid processing empty Fields so that the absence of required components in an empty Field is not treated as an error.

Return Value

Returns 1 if the Field is empty (see above); otherwise, returns 0.

See Also

DXExists

“Standard Components” on page 107.

DXEndField

Function

Creates the standard components “box” and “neighbors” that other modules expect Field **f** to contain.

Syntax

```
#include <dx/dx.h>

Field DXEndField(Field f)
```

Functional Details

The “box” component defines the corners of an n-dimensional box that contains all of the positions described by the Field’s “positions” component. The “neighbors” component contains information that provides a mechanism for quick access to neighboring interpolation elements.

Note: A “neighbors” component is not created for a Field with regular connections.

In addition, **DXEndField** sets the “dep” and “ref” attributes, if not already set, on the components listed in Table 4 on page 225.

Table 4. Set Attributes

Component	Attribute	Value
“positions”	“dep”	“positions”
“connections”	“ref”	“positions”
“data”	“dep”	“positions”
“colors”	“dep”	“positions”
“front colors”	“dep”	“positions”
“back colors”	“dep”	“positions”
“opacities”	“dep”	“positions”
“tangents”	“dep”	“positions”
“normals”	“dep”	“positions”
“binormals”	“dep”	“positions”

During this phase, **DXEndField** also checks to make sure that the number of elements in a component being set to depend on the “positions” component does actually match the number of positions in the Field.

DXEndField also trims all of a Field’s component Arrays to use the amount of space actually containing data as specified by **DXAddArrayData**. Thus, for optimal performance, **DXEndField** should be called just prior to returning a Field. Further, after a call to **DXEndField**, pointers obtained by calls to **DXGetArrayData** to data contained in Arrays that are components of a Field cannot be assumed valid.

If you are creating a Group that contains several Fields, then **DXEndObject** may be called instead on the Group structure. This will parallelize the application of the **DXEndField** calls to the member Fields on architectures that support parallelism.

For a more complete examination of a Field, including checks to ensure that indices contained in components that refer to other components are correct, see additional details of the **Verify** module in Chapter 1, “Data Explorer Tools” on page 1 in *IBM Visualization Data Explorer User’s Reference*.

Return Value

Returns **f** or returns **NULL** and sets an error code.

See Also

DXBoundingBox, DXChangedComponentStructure, DXChangedComponentValues, DXEmptyField, DXEndObject, DXNeighbors, DXStatistics, Verify

“Standard Components” on page 107.

DXEndObject

Function

Creates the standard components “box” and “neighbors” that other modules expect a Field to contain.

Syntax

```
#include <dx/dx.h>

Object DXEndObject(Object o)
```

Functional Details

DXEndObject provides a higher-level interface to the Field-completion processing provided by **DXEndField**, since it will traverse a variety of Object classes to access embedded Fields. In addition, **DXEndObject** detects if the Fields contained as subObjects of **o** share components, and if so, share the newly created “box” and/or “neighbors” components between Fields containing the shared components. Finally, **DXEndObject** processes the Fields that are subObjects of **o** in parallel on architectures supporting parallelism.

Thus, unless the Object to be completed is simply a single Field, **DXEndObject** provides a more flexible, faster, and potentially more space-efficient processing stage than **DXEndField**.

Return Value

Returns **o** or returns **NULL** and sets an error code.

See Also

DXEndField

“Standard Components” on page 107.

DXExecuteTaskGroup

Function

Runs the group of tasks in the current group in parallel, if possible.

Syntax

```
#include <dx/dx.h>

Error DXExecuteTaskGroup()
```

Functional Details

Begins executing the tasks belonging to the current task group. **DXCreateTaskGroup** waits for the completion of all tasks in this task group. The tasks are started in decreasing order of the work estimate given in **DXAddTask**.

Return Value

Returns **OK** if all tasks in the task group complete without error; otherwise, returns **ERROR**. Any error code returned is set by the task involved.

See Also

DXAddTask, DXCreateTaskGroup

12.8, "Parallelism" on page 123.

DXExists

Function

Determines if a component exists in a Field.

Syntax

```
#include <dx/dx.h>

Object DXExists(Object o, char *name)
```

Functional Details

If any Field in Object **o** contains a component of the specified **name**, this routine returns **o**. Object **o** can be a single Field or any Object that can contains Fields (e.g., Groups or Series).

Return Value

Returns **o** if any Field in Object **o** contains a **name** component; otherwise, **NULL** but does not set an error code.

See Also

DXExtract, DXGetComponentValue, DXInsert, DXRemove DXRename, DXReplace, DXSwap

11.10, "Component Manipulation" on page 110.

DXExportDX

Function

Writes an Object to a specified file in a specified Data Explorer format.

Syntax

```
#include <dx/dx.h>

Error DXExportDX(Object o, char *filename, char *format)
```

Functional Details

Writes the contents of Object **o** into the file specified by **filename** using format **format**. **format** must be "dx" with the following optional keywords appended:

Table 5. Format Keyword Description

Keyword	Format of the Data in the File
text	data in ASCII
ASCII	data in ASCII
binary	data in binary
1	header and data in a single file: header section, then data section
2	header and data in two files
follows	header and data in a single file; data immediately following object description in header section

You may specify more than one keyword (e.g., **format**="dx text 2").

Return Value

Returns **OK** or returns **NULL** and sets an error code.

See Also

DXImportDX

"Data Explorer Format Files" on page 110.

DXExtract

Function

Extracts a component from a Field.

Syntax

```
#include <dx/dx.h>

Object DXExtract(Object o, char *name)
```

Functional Details

For each Field in Object **o**, the routine returns the Object specified by **name** (typically an Array). Object **o** can be a simple Field or any Object that can contain Fields (e.g., Groups or Series).

If Object **o** is a single Field, a single Object is returned (typically an Array). If Object **o** is anything else, the Object hierarchy is preserved, and each Field is replaced by component **name**.

Return Value

Returns **o** or returns **NULL** and sets an error code. It is an error if no component of the specified **name** is found in any Field of **o**.

See Also

DXExists, DXGetComponentValue, DXInsert, DXRemove, DXRename, DXReplace, DXSwap

11.10, "Component Manipulation" on page 110.

DXExtractFloat

Function

Determines whether an Object can be converted to a floating-point value, and if so, extracts it.

Syntax

```
#include <dx/dx.h>

Object DXExtractFloat(Object o, float *fp)
```

Functional Details

If Object **o** can be converted to a floating-point value (i.e., **TYPE_FLOAT**, **CATEGORY_REAL**, rank 0), this routine extracts it and places it in ***fp**.

Return Value

Returns **o** and sets ***fp** if a floating-point value can be extracted from **o**; otherwise, returns **NULL** without setting an error code.

See Also

DXExtractInteger, DXExtractNthString, DXExtractParameter, DXExtractString, DXQueryArrayConvert, DXQueryParameter

11.8, "Extracting Module Parameters" on page 108.

DXExtractInteger

Function

Determines whether an Object can be converted to an integer, and if so, extracts it.

Syntax

```
#include <dx/dx.h>

Object DXExtractInteger(Object o, int *ip)
```

Functional Details

If Object **o** can be converted into an integer (i.e., **TYPE_INT**, **CATEGORY_REAL**, rank 0), this routine extracts the integer value and places it in ***ip**.

Return Value

Returns **o** and sets **ip** if an integer can be extracted from **o**; otherwise, returns **NULL** without setting an error code.

See Also

DXExtractFloat, DXExtractNthString DXExtractParameter, DXExtractString, DXQueryArrayConvert, DXQueryParameter

11.8, “Extracting Module Parameters” on page 108.

DXExtractNthString

Function

Determines whether an Object can be converted to a list of strings and, if so, extracts the **n**th one from it.

Syntax

```
#include <dx/dx.h>

Object DXExtractNthString(Object o, int n, char **cp)
```

Functional Details

If Object **o** contains at least an **n**th string (where **n** is a zero-based index), this routine extracts it and places a pointer to its first character in ***cp**.

Strings may be extracted from:

- String Objects (if **n** is 0)
- String lists (e.g., those created by **DXMakeStringList** and **DXMakeStringListV**).

Return Value

Returns **o** and sets **cp** if the **n**th string can be extracted from **o**; otherwise, returns **NULL** without setting an error code.

See Also

DXExtractFloat, DXExtractInteger, DXExtractParameter, DXExtractString, DXMakeStringList, DXMakeStringListV, DXQueryArrayConvert, DXQueryParameter

11.8, "Extracting Module Parameters" on page 108.

DXExtractParameter

Function

Determines whether an Object can be converted to a specific value type and, if so, returns the value in the user-provided buffer.

Syntax

```
#include <dx/dx.h>

Object DXExtractParameter(Object o, Type t, int dim, int count, Pointer p)
```

Functional Details

If Object **o** can be converted to Type **t** with dimensionality **dim** and **count** elements, this routine performs the conversion and returns the data in the buffer pointed to by **p**.

For a successful conversion, Object **o** must be either an Array or a String. If **o** is an Array, then its Category must be **CATEGORY_REAL**, its rank must be either 0 or 1, and it must have **count** items contained within.

If **dim** is greater than 1, then **o**'s rank must be 1 and its shape must match **dim** in order for this conversion to be successful. If **dim** is either 0 or 1, then both rank 0 and rank 1 shape 1 Arrays will match in size.

Once it is known that the sizes match, the Array's Type is examined to determine whether it can be converted to the Type specified by **t**. In general, smaller signed (or unsigned) types can be converted to larger signed (or unsigned) types as follows:

TYPE_BYTE ⇒ **TYPE_SHORT** ⇒ **TYPE_INT** ⇒ **TYPE_FLOAT** ⇒ **TYPE_DOUBLE**

and

TYPE_UBYTE ⇒ **TYPE_USHORT** ⇒ **TYPE_UINT**

Signed and unsigned versions of the same type cannot be converted between each other (e.g., **TYPE_BYTE** and **TYPE_UBYTE**). However, unsigned types can be converted to larger signed types (e.g., **TYPE_UBYTE** ⇒ **TYPE_SHORT**).

If the Types are identical, the data contained in **o** is copied to the buffer pointed to by **p**. If (without violating any of the rules just given) **o** can be converted to the Type specified in **t**, it is, and the converted data is copied to the buffer (**p**).

If **o** is a String, then **t** must be **TYPE_STRING** and **dim** must be either 0 or 1. If **dim** is 0, then the string contained in **o** must consist only of a single character.

Return Value

Returns **o** if the conversion is valid; otherwise, returns **NULL** without setting an error code.

See Also

DXExtractFloat, DXExtractInteger, DXExtractNthString, DXExtractString, DXQueryArrayConvert, DXQueryParameter

11.8, "Extracting Module Parameters" on page 108.

DXExtractString

Function

Determines whether an Object can be converted to a String, and if so, extracts it.

Syntax

```
#include <dx/dx.h>

Object DXExtractString(Object o, char **cp)
```

Functional Details

If Object **o** contains a string, the routine extracts it and places a pointer to its first character in ***cp**. Strings may be extracted from any of the following:

- String Objects
- String lists containing a single element, such as those created by **DXMakeStringList** and **DXMakeStringListV**
- Array Objects containing data of Type **TYPE_UBYTE** with a single **NULL** at the end and no embedded **NULLS**. This is for backward compatibility only; its use is *not* recommended.

Return Value

Returns **o** and sets **cp** if a string can be extracted from **o**; otherwise, returns **NULL** without setting an error code.

See Also

DXExtractFloat, DXExtractInteger, DXExtractNthString, DXExtractParameter, DXMakeStringList, DXMakeStringListV, DXQueryArrayConvert, DXQueryParameter

11.8, "Extracting Module Parameters" on page 108.

DXFree

Function

Frees a previously allocated block of memory.

Syntax

```
#include <dx/dx.h>

Error DXFree(Pointer x)
```

Functional Details

This routine can be used to free memory (pointed to by **x**) that has been allocated by any of the following routines: **DXAllocate, DXAllocateZero, DXAllocateLocal, DXAllocateLocalZero,** or **DXReAllocate**. If **x** is **NULL**, **DXFree** immediately returns **OK**.

Return Value

Returns **OK** or returns **ERROR** and sets an error code.

See Also

DXAllocate, DXAllocateLocal, DXAllocateLocalZero, DXAllocateZero, DXPrintAlloc, DXReAllocate

12.3, “Memory Allocation” on page 116.

DXFreeArrayDataLocal

Function

Frees space previously allocated by **DXGetArrayDataLocal**.

Syntax

```
#include <dx/dx.h>

Array DXFreeArrayDataLocal(Array a, Pointer data)
```

Functional Details

This routine must be called from the same task that called **DXGetArrayDataLocal** if the program is running in parallel. If called on a machine without processor local memory, it simply returns without setting an error code.

Return Value

Returns **a** or returns **NULL** and sets an error code.

See Also

DXGetArrayData, DXGetArrayDataLocal

11.3, “Array Class” on page 101.

DXFreeArrayHandle

Function

Frees the memory allocated for an Array handle.

Syntax

```
#include <dx/dx.h>

Error DXFreeArrayHandle(ArrayHandle handle)
```

Functional Details

Frees the memory allocated for the Array handle **handle**.

Return Value

Returns **OK** or returns **ERROR** and sets an error code.

See Also

DXCreateArrayHandle

“Array Handling” on page 102.

DXFreeInvalidComponentHandle

Function

Frees all memory associated with an invalid-component handle.

Syntax

```
#include <dx/dx.h>

Error DXFreeInvalidComponentHandle(InvalidComponentHandle handle)
```

Functional Details

Frees all memory associated with the invalid-component handle **handle**.

Return Value

Returns **OK** or returns **ERROR** and sets an error code.

See Also

DXCreateInvalidComponentHandle, DXFreeInvalidComponentHandle

13.3, “Invalid Data” on page 133.

DXFreeModuleID

Function

Frees the space pointed to by a module identifier.

Syntax

```
#include <dx/dx.h>

Error DXFreeFreeModuleID(Pointer id)
```

Functional Details

If DXCopyModuleID was used to obtain a copy of a module identifier, DXFreeModuleID should be used to delete that copy when it is no longer needed.

Return Value

Returns **OK** or returns **ERROR** and sets an error code.

See Also

13.3, “Invalid Data” on page 133

12.5, “Cache” on page 121

12.11, “Asynchronous Services” on page 129.

DXGeometricText

Function

Produces a geometric text Object consisting of a given string.

Syntax

```
#include <dx/dx.h>

Object DXGeometricText(char *s, Object font, float *width)
```

Functional Details

The geometric text Object consists of the string **s**, in the font **font** (which is an Object returned by **DXGetFont**. The text is a Field with a "positions" component indicating the points of the font in pixels, a "connections" component of type "lines," and a "colors" component with a constant white color. The origin of the string (left end of baseline) is placed at the origin of the *x*,*y* plane with the baseline pointed along the positive *x* axis. If **width** is not **NULL**, the width of the string (in pixels) will be returned in ***width**. The string will be bounded above by **ascent** and below by **-descent** (as returned by **DXGetFont**), to the left by 0, and to the right by **width**.

Return Value

Returns the text Object or returns **NULL** and sets an error code.

See Also

DXGetFont

14.1, "Text" on page 146.

DXGetArrayClass

Function

Returns the subclass of an Array Object.

Syntax

```
#include <dx/dx.h>

Class DXGetArrayClass(Array a)
```

Functional Details

Returns **CLASS_ARRAY** if the Array **a** is irregular. Otherwise returns one of **CLASS_REGULARARRAY**, **CLASS_PRODUCTARRAY**, **CLASS_PATHARRAY**, **CLASS_MESHARRAY,** or **CLASS_CONSTANTARRAY**.

Return Value

Returns the subclass of an Array Object.

See Also

`DXGetArrayData, DXGetArrayInfo`

11.3, “Array Class” on page 101.

DXGetArrayData

Function

Returns a pointer to the start of a global memory area containing the items constituting the data stored in an Array.

Syntax

```
#include <dx/dx.h>

Pointer DXGetArrayData(Array a)
```

Functional Details

For irregular Arrays, the pointer points to the actual data that was stored in the Array; this data may be changed directly to change the contents of the Array. For compact Arrays (regular, grid, path, or mesh Arrays), this routine expands the compact data and returns a pointer to the result; such data should not be changed because changes to this data will not be reflected in the original Array. The returned Array contains *n* items numbered from *0* to *n-1*, where *n* is the number of items in **`a`**. **`DXAddArrayData`** must be called before calling **`DXGetArrayData`**; otherwise, the values of items in an irregular Array are undefined.

Note: To reduce memory requirements, it is preferable, where possible, to recognize compact Arrays using **`DXGetArrayClass`**, and not to expand them by calling **`DXGetArrayData`**. The Array handle routines may be used to access arrays of any class without expansion.

Memory pointed to by the return from **`DXGetArrayData`** should *not* be freed by the user.

Return Value

Returns a pointer to the data or returns **`NULL`** and sets an error code.

See Also

`DXAddArrayData, DXCreateArrayHandle, DXGetArrayClass, DXGetArrayDataLocal, DXNewArray, DXNewArrayV`

11.3, “Array Class” on page 101.

DXGetArrayDataLocal

Function

Returns a pointer to the start of memory of a local copy of the data stored in an Array.

Syntax

```
#include <dx/dx.h>

Pointer DXGetArrayDataLocal(Array a)
```

Functional Details

This routine performs the same operation as **DXGetArrayData**, on a machine without processor local memory.

On a machine with processor local memory, it performs the same operation as **DXGetArrayData**, after which, the Array data contents of **a** are copied to local memory. When you no longer need the local copy, **DXFreeArrayDataLocal** must be called.

The local data should be considered a read-only copy.

Return Value

Returns a pointer to the data or returns **NULL** and sets an error code.

See Also

DXFreeArrayDataLocal, DXGetArrayData

11.3, “Array Class” on page 101.

DXGetArrayEntry, DXGetArrayEntries

Function

Return a specified item or items from an Array.

Syntax

```
#include <dx/dx.h>

Pointer DXGetArrayEntry(ArrayHandle handle, int offset, Pointer scratch)

void DXGetArrayEntries(ArrayHandle handle, int count, int *offsets,
                       Pointer *entries, Pointer scratch)
```

Functional Details

Given an **offset** or list of ***offsets** into an Array, the routine returns a pointer or pointers to the memory location(s) containing the Array elements specified. (The Array is specified by the Array handle **handle**, which must first be created by **DXCreateArrayHandle**).

For **DXGetArrayEntry**: The region of memory pointed to by **scratch** must be large enough to hold a specified element.

For **DXGetArrayEntries**: The routine returns a list of pointers in **entries**, which must be large enough to hold **count** pointers. The parameter **scratch** must be large enough to hold **count** items of the Array.

Return Value

Return a pointer or pointers to the specified entry or entries if the "offset" value(s) are valid for the Array. If not, the results are undefined. Note that you should use the return value of this function, not **`scratch`**.

See Also

`DXCreateArrayHandle, DXFreeArrayHandle, DXGetArrayEntry, DXGetArrayEntries, DXIterateArray`

"Array Handling" on page 102.

DXGetArrayInfo

Function

Returns the number of items, type, category, rank, and shape of an Array.

Syntax

```
#include <dx/dx.h>

Array DXGetArrayInfo(Array a, int *items, Type *type, Category *category,
                     int *rank, int *shape)
```

Functional Details

If **`items`** is not **`NULL`**, this routine returns in **`*items`** the number of items currently in the Array. If type is not **`NULL`**, it returns in **`*type`** the type of each item. If **`category`** is not **`NULL`**, it returns in **`*category`** the category of each item. If **`rank`** is not **`NULL`**, it returns in **`*rank`** the number of dimensions in each item. If shape is not **`NULL`**, it returns in **`*shape`** an Array of the extents of each dimension of the items.

Library Routines

The type is one of the following:

`TYPE_BYTE`	**`TYPE_HYPER`**	**`TYPE_SHORT`**
`TYPE_UBYTE`	**`TYPE_INT`**	**`TYPE_USHORT`**
`TYPE_DOUBLE`	**`TYPE_UINT`**	**`TYPE_STRING`**
`TYPE_FLOAT`		

The category is either **`CATEGORY_REAL`** or **`CATEGORY_COMPLEX`**.

(For information on rank and shape, see Chapter 3, "Understanding the Data Model" on page 15 in *IBM Visualization Data Explorer User's Guide*.)

Return Value

Returns **`a`** or returns **`NULL`** without setting an error code.

See Also

`DXGetArrayClass, DXGetItemSize, DXTypeCheck, DXTypeCheckV`

11.3, "Array Class" on page 101.

DXGetAttribute

Function

Retrieves a named attribute from an Object.

Syntax

```
#include <dx/dx.h>

Object DXGetAttribute(Object o, char *name)
```

Functional Details

Retrieves the attribute specified by the string **name** from Object **o**.

Return Value

Returns the Object associated with the attribute name or returns **NULL** without setting an error code.

See Also

DXGetEnumeratedAttribute, DXGetFloatAttribute, DXGetIntegerAttribute, DXGetStringAttribute, DXSetAttribute, DXDeleteAttribute, DXSetFloatAttribute, DXSetIntegerAttribute, DXSetStringAttribute

"Object Routines" on page 119.

DXGetCacheEntry, DXGetCacheEntryV

Function

Retrieve a cache entry.

Syntax

```
#include <dx/dx.h>

Object DXGetCacheEntry(char *function, int key, int n, ...)
Object DXGetCacheEntryV(char *function, int key, int n, Object *in)
```

Functional Details

Both routines return the Object referenced by a cache entry. The cache is indexed by a key created from **function**, **key**, **n**, and the Objects in the Array **in**. These must be the same values as those used when the Object was placed in the cache with **DXSetCacheEntry**.

Notes:

1. Because Data Explorer modules follow pure function semantics, the cache should *not* be used to store a state that affects the output of the module. A module must always be able to recreate the Object from the same set of inputs; the cache should only be used as an optimization tool.
2. On a multiprocessor machine, processor local information should *not* be stored in the cache, since its contents may be retrieved on another processor.

3. The cache is local to one machine and cannot be used to communicate information between modules on different machines when running in distributed mode.

Since DXGetCacheEntry returns an Object that is referenced so that it will not be deleted, you must delete it when you are finished with it. Failure to do so will result in a memory leak.

For additional details on the deletion of cached Objects, see **DXSetCacheEntry**.

Return Value

Returns the cached output Object or returns **NULL** but does not set an error code if no such cache entry exists.

See Also

DXDelete, DXReference, DXSetCacheEntry, DXSetCacheEntryV, DXGetObjectTag, DXSetObjectTag

12.5, "Cache" on page 121.

DXGetCameraMatrix, DXGetCameraRotation, DXGetCameraMatrixWithFuzz

Function

Return matrices that represent stages of the viewing operation.

Syntax

```
#include <dx/dx.h>

Matrix DXGetCameraMatrix(Camera c)
Matrix DXGetCameraRotation(Camera c)
Matrix DXGetCameraMatrixWithFuzz(Camera c, float fuzz)
```

Functional Details

A camera defines the position and orientation of the viewer, the volume of interest of the object being viewed, and the size of the image to contain the resulting view.

Conceptually, there are three steps to converting an object in 3-dimensional space to an image in two-dimensional space:

1. Translate the Object and camera as a unit so that **From** is at the center of the image, rotate them until **up** is aligned with the image **y** axis, and the **from-to** vector is perpendicular to the image.

   DXGetCameraRotation, given camera **c**, returns the matrix that performs this series of transformations.

2. Scale the size of the volume of interest to the size of the image. (For additional details, see **DXSetPerspective** and **DXSetOrthographic**.)

   DXGetCameraMatrix and **DXGetCameraMatrixWithFuzz**, given camera **c**, return the matrices that perform this step and the previous step. **DXGetCameraMatrixWithFuzz** returns the matrix with a given **fuzz** applied. The **fuzz** allows an object that is coincident with a second object, to be moved

Library Routines

forward slightly to allow it to appear in front of the second object. An example of this would be displaying contour lines on a surface; a positive **fuzz** applied to the lines make the lines appear displayed slightly in front of the surface.

3. For perspective cameras only, make the walls of the volume interest parallel. (For additional details, see **DXSetPerspective**.)

When these three steps are completed, the object has the correct image x,y coordinates and can be rasterized.

Return Value

Return the matrix.

See Also

DXSetResolution, DXGetCameraResolution, DXSetView, DXGetView, DXNewCamera, DXSetOrthographic, DXGetOrthographic, DXSetPerspective, DXGetPerspective, DXRender

15.7, "Camera Class" on page 155.

DXGetClippedInfo

Function

Returns the Object to be rendered and the clipping Object.

Syntax

```
#include <dx/dx.h>

Clipped DXGetClippedInfo(Clipped c, Object *render, Object *clipping)
```

Functional Details

Returns the Object to be rendered in ***render**, given a Clipped Object **c**, if **render** is not **NULL**. If **clipping** is not **NULL**, returns the clipping Object in ***clipping**.

Return Value

Returns **c** or returns **NULL** and sets an error code.

See Also

DXNewClipped, DXSetClippedObjects

15.6, "Clipped Class" on page 155.

DXGetComponentAttribute

Function

Returns a named attribute of a specified component of a Field.

Syntax

```
#include <dx/dx.h>

Object DXGetComponentAttribute(Field f, char *name, char *attribute)
```

Functional Details

The routine first retrieves the component **name** from Field **f**. If the component exists, the attribute specified by the string **attribute** associated with that component is returned.

Return Value

Returns the Object associated with **attribute** of component **name** or returns **NULL** without setting an error code (unless **f** is not a Field).

See Also

DXGetAttribute, DXGetComponentValue, DXGetEnumeratedComponentAttribute, DXGetFloatAttribute, DXGetIntegerAttribute, DXSetComponentAttribute

11.1, “Field Class” on page 97.

DXGetComponentValue

Function

Returns a specified component of a specified Field.

Syntax

```
#include <dx/dx.h>

Object DXGetComponentValue(Field f, char *name)
```

Functional Details

Typically, Fields have a “positions” component defining a set of points in space, a “connections” component defining the connectivity of the positions, and a “data” component containing the individual data values associated with the individual positions or connections. A Field can be manipulated by accessing these components. The following are just two examples:

- You can transform a Field by calling **DXGetComponentValue(field, “positions”)** to return the “positions” component, and then transforming the points it contains.
- You can convert a Field containing vector data to one containing the magnitude of the vector data by calling **DXGetComponentValue(field, “data”)** to return the “data” component, and then creating a new “data” component containing the magnitude data to replace it in the Field.

Return Value

Returns the **name** component of Field **f** or returns **NULL** and Sets an error code if **f** is not a Field. It does not set an error code if the component does not exist.

See Also

`DXDeleteComponent`, `DXGetComponentAttribute`, `DXGetEnumeratedComponentValue`, `DXSetComponentValue`

11.1, “Field Class” on page 97.

DXGetConnections

Function

Gets the “connections” component of a Field and checks to see if it has a specified “element type” attribute.

Syntax

```
#include <dx/dx.h>

Array DXGetConnections(Field f, char *type)
```

Functional Details

This routine combines the functions of **`DXGetComponentValue`** and **`DXGetComponentAttribute`**.

Return Value

Returns the connections Array or returns **`NULL`** without setting an error code if (1) no “connections” component is present or (2) if the “element type” attribute does not match.

See Also

`DXGetComponentAttribute`, `DXGetComponentValue`, `DXSetConnections`

“Connections” on page 107.

DXGetConstantArrayData

Function

Returns a pointer to the value stored in a Constant Array.

Syntax

```
#include <dx/dx.h>

Pointer DXGetConstantArrayData (Array a)
```

Functional Details

The data pointed to by this pointer should be interpreted by the user according the type, category, rank, and shape parameters associated with **`a`**.

While a Constant Array may contain numerous items, as may be indicated by either **`DXGetArrayInfo`** or **`DXQueryConstantArray`**, the pointer returned by **`DXGetConstantArrayData`** should not be incremented beyond the amount necessary to index a single item since the value of all items in a Constant Array is actually stored in a single item.

For compatibility with previous versions, this routine will also work with Regular Arrays where the delta vectors consist solely of zeros.

Return Value

Returns a pointer to the data contained in the Constant Array or returns **NULL** and sets an error code.

See Also

DXGetArrayData, DXGetArrayInfo, DXNewConstantArray, DXNewConstantArrayV, DXQueryConstantArray

“Constant Arrays” on page 105.

DXGetEnumeratedAttribute

Function

Retrieves the **n**th attribute from an Object.

Syntax

```
#include <dx/dx.h>

Object DXGetEnumeratedAttribute(Object o, int n, char **name)
```

Functional Details

The attribute to be retrieved from Object **o** is specified by the zero-based index **n**. If **o** has an **n**th attribute, and **name** is not **NULL**, the string associated with the **n**th attribute will be returned in **name**.

Return Value

Returns the Object associated with the index **n** or returns **NULL** without setting an error code.

See Also

DXGetAttribute, DXGetFloatAttribute, DXGetIntegerAttribute, DXGetStringAttribute, DXSetAttribute, DXSetFloatAttribute, DXSetIntegerAttribute, DXSetStringAttribute

“Object Routines” on page 119.

DXGetEnumeratedComponentAttribute

Function

Provides access to a specified attribute of a component by index rather than by name.

Syntax

```
#include <dx/dx.h>

Object DXGetEnumeratedComponentAttribute(Field f, int n, char **name, char *attribute)
```

Functional Details

The **attribute** attribute of the **n**th component of Field **f** is returned. The name of the component is returned in ***name**.

This interface is used primarily to provide access to all components of the Field **f** without requiring a list of component names.

Return Value

Returns the attribute or returns **NULL** but does not set an error code if **n** is out of range.

See Also

DXGetComponentAttribute, DXGetEnumeratedComponentValue

11.1, “Field Class” on page 97.

DXGetEnumeratedComponentValue

Function

Provides access to the components of a Field by index rather than by name.

Syntax

```
#include <dx/dx.h>

Object DXGetEnumeratedComponentValue(Field f, int n, char **name)
```

Functional Details

Provides access to all components of the Field **f** without requiring a list of component names. We might, for example, find all components that are dependent on positions by using **DXGetEnumeratedComponentValue** in a looping construct in which **DXGetEnumeratedComponentAttribute** (or, using the name returned by **DXGetEnumeratedComponentValue**, using **DXGetComponentAttribute**) is used to access the “dep” attribute of each component.

The components of Field **f** may be indexed by calling **DXGetEnumeratedComponentValue** with successive values of **n** until **NULL** is returned. The name of the component is returned in ***name**.

Note: **DXGetEnumeratedComponentAttribute** would not be suitable for use in the looping construct because it will return **NULL** if the **n**th component doesn't have the specified component, even if there are more than **n** components.

Return Value

Returns the value of the component or returns **NULL** but does not set an error code if **n** is out of range.

See Also

DXGetComponentAttribute, DXGetComponentValue, DXGetEnumeratedComponentAttribute

11.1, "Field Class" on page 97.

DXGetEnumeratedMember

Function

Returns the members of a Group by index.

Syntax

```
#include <dx/dx.h>

Object DXGetEnumeratedMember(Group g, int n, char **name)
```

Functional Details

Returns the **n**th member of Group **g**. The members of a Group may be indexed by calling this routine with successive values of **n** starting with 0 until **NULL** is returned. This routine returns the name of the **n**th member in ***name** if **name** is not **NULL**.

Note: The numbering changes as members are added and deleted.

Return Value

Returns the **n**th member or returns **NULL** but does not set an error code if **n** is out of range. Sets an error code if **g** is not a Group.

See Also

DXGetMember, DXGetMemberCount, DXNewGroup, DXSetEnumeratedMember

"Generic Operations" on page 98.

DXGetError

Function

Returns the error code for the last error that occurred.

Syntax

```
#include <dx/dx.h>

ErrorCode DXGetError()
```

Functional Details

Returns the error code for the last error that occurred, which is one of the following error codes:

```
ERROR_ASSERTION       ERROR_INTERNAL       ERROR_NO_MEMORY
ERROR_BAD_CLASS       ERROR_INVALID_DATA   ERROR_NO_NONE
ERROR_BAD_PARAMETER   ERROR_MISSING_DATA   ERROR_NOT_IMPLEMENTED
ERROR_BAD_TYPE        ERROR_NO_CAMERA      ERROR_UNEXPECTED
```

A return value of **ERROR_NONE** signifies that no error code has been set.

Note: This routine is not typically used by module writers.

Return Value

Returns the error code for the most recent error or returns **ERROR_NONE** if no error code has been set.

See Also

DXGetErrorMessage, DXResetError, DXSetError

12.1, "Error Handling and Messages" on page 114.

DXGetErrorExit

Function

Returns the current value of **level** as set by **DXSetErrorExit**.

Syntax

```
#include <dx/dx.h>

int DXGetErrorExit();
```

Functional Details

This routine is intended for use only in stand-alone programs.

Return Value

Returns 0, 1, or 2: Valid arguments for **level** are:

0 = store error message. Use **DXPrintError()** to print when ready.

1 = print error message and return.

2 = print error message and exit.

See Also

DXSetErrorExit

12.10, "Module Access" on page 127.

DXGetErrorMessage

Function

Returns the current error message.

Syntax

```
#include <dx/dx.h>

char *DXGetErrorMessage()
```

Functional Details

Returns a pointer to the current error message **NULL**-terminated string. This is a pointer to a static buffer in local memory, so it must be copied if it is to be used outside the scope of the calling routine.

Return Value

Returns a pointer to a null string if the error code is **ERROR_NONE**. Otherwise, returns a pointer to the current error message.

See Also

DXSetErrorExit

12.1, "Error Handling and Messages" on page 114.

DXGetFloatAttribute

Function

Retrieves a named attribute from an Object, verifies that it contains a floating-point number, and returns that number.

Syntax

```
#include <dx/dx.h>

Object DXGetFloatAttribute(Object o, char *name, float *x)
```

Functional Details

The attribute to be retrieved from Object **o** is specified by the string **name**. The routine then verifies that the attribute contains a scalar floating-point value. If **x** is not **NULL**, the floating-point value is returned in ***x**.

Return Value

Returns **o** or returns **NULL** without setting an error code.

See Also

DXGetAttribute, **DXGetEnumeratedAttribute**, **DXGetIntegerAttribute**, **DXGetStringAttribute**, **DXSetAttribute**, **DXSetFloatAttribute**, **DXSetIntegerAttribute**, **DXSetStringAttribute**

"Object Routines" on page 119.

DXGetFont

Function

Returns a Group representing the named font.

Syntax

```
#include <dx/dx.h>

Object DXGetFont(char *name, float *ascent, float *descent)
```

Functional Details

The Group has as many members as there are characters in the named font (**name**). Typically, the returned font is passed to **DXGeometricText** to construct a text Object from a given string. Alternatively, individual characters can be extracted using **DXGetEnumeratedMember**. The member number is the same as the ASCII character code.

The font has an overall height of 1. If **ascent** is not **NULL**, the portion of the overall height above the baseline is returned in ***ascent**. If **descent** is not **NULL**, the portion of the overall height below the baseline is returned in ***descent**. The sum of the ascent and the descent is the overall height 1.

This routine checks the environment variables DXFONTS, DXEXECROOT, and DXROOT for directories to search for fonts. The routine also checks in `/usr/lpp/dx` and in the subdirectory **fonts** of each of these directories.

Return Value

Returns the font or returns **NULL** and sets an error code.

See Also

DXGeometricText, DXGetEnumeratedMember

14.1, “Text” on page 146.

DXGetGroupClass

Function

Returns the subclass of a Group Object.

Syntax

```
#include <dx/dx.h>

Class DXGetGroupClass(Group g)
```

Functional Details

Returns the subclass of a Group Object **g**. This will be **CLASS_GROUP** if the Object is a generic Group Object, or either **CLASS_SERIES**, **CLASS_MULTIGRID**, or **CLASS_COMPOSITEFIELD** if the Object class is a subclass of Group.

A **CLASS_GROUP** Object and its subclasses contain other Objects that are referred to as members. The members of a **CLASS_GROUP** Object can be of any class and are not restricted as to type, whereas the **CLASS_SERIES**, **CLASS_MULTIGRID**, and **CLASS_COMPOSITEFIELD** generally contain members of **TYPE_FIELD,** and they assume the type of the first typed member to be added and are untyped if empty. All of the subclasses require that the type of all its members is the same; see **DXSetGroupType**. A **CLASS_SERIES** is generally used to maintain time-variant data, whereas the **CLASS_COMPOSITEFIELD** and **CLASS_MULTIGRID** are used to maintain spatially partitioned data within a single field.

Note: A Group can be structured to take advantage of more than one Group class. For example, one could have a Series of Composite Fields where each Series member is a Field that has been partitioned into a Composite Field.

Return Value

Returns the subclass of a Group Object or returns an undefined value if **g** is not a Group. The subclass returned will be:

- **CLASS_GROUP**, if the Object is a generic Group Object; or
- **CLASS_SERIES**, **CLASS_MULTIGRID**, or **CLASS_COMPOSITEFIELD**, if the Object class is a subclass of **Group**.

See Also

DXGetObjectClass, DXNewCompositeField, DXNewGroup, DXNewMultiGrid, DXNewSeries

“Generic Operations” on page 98.

DXGetImageSize, DXGetImageBounds

Function

Return information about image Fields.

Syntax

```
#include <dx/dx.h>

Field DXGetImageSize(Field i, int *width, int *height)
Object DXGetImageBounds(Object o, int *x, int *y, int *width, int *height)
```

Functional Details

DXGetImageSize returns the **width** and **height** of a simple image Field **i**.

DXGetImageBounds returns the origin and dimensions of a simple or composite image Field **o** (such as is generated by the Arrange module). The origin is the offset of this part of the image compared to the whole image.

Return Value

Return the image or return **NULL** and sets an error code.

See Also

DXGetPixels, DXMakeImage

15.9, "Image Fields" on page 156.

DXGetIntegerAttribute

Function

Retrieves a named attribute from an Object, verifies that its contents are an integer number, and returns that number.

Syntax

```
#include <dx/dx.h>

Object DXGetIntegerAttribute(Object o, char *name, int *x)
```

Functional Details

Retrieves the attribute specified by the string **name** from Object **o**. It then verifies that the attribute contains a scalar integer value. If **x** is not **NULL**, the integer value is returned in ***x**.

Return Value

Returns **o** or returns **NULL** without setting an error code.

See Also

DXGetAttribute, DXGetEnumeratedAttribute, DXGetFloatAttribute, DXGetStringAttribute, DXSetAttribute, DXSetFloatAttribute, DXSetIntegerAttribute, DXSetStringAttribute

"Object Routines" on page 119.

DXGetInvalidComponentArray

Function

Returns an Array containing the information stored in an invalid-component handle.

Syntax

```
#include <dx/dx.h>

Array DXGetInvalidComponentArray(InvalidComponentHandle handle)
```

Functional Details

The returned Array may be dependent or referential (see Chapter 13, “Data Processing” on page 131). If **handle** was created using a Field, then the Array will contain the appropriate “dep” or “ref” attributes. However, if **handle** was created using only an Array, then this routine cannot determine whether the returned Array is dependent or it references “positions” or “connections,” and it is up to the calling program to set the appropriate attribute. The determination of should be based on the type of the Array; if it is **TYPE_UBYTE** or **TYPE_BYTE**, it is dependent; if it is **TYPE_UINT** or **TYPE_INT**, it is referential.

Return Value

Returns the Array or returns **NULL** and sets an error code.

See Also

DXCreateInvalidComponentHandle, DXFreeInvalidComponentHandle, DXGetType

13.3, “Invalid Data” on page 133.

DXGetInvalidCount

Function

Returns the number of invalid elements in an invalid-component handle.

Syntax

```
#include <dx/dx.h>

int DXGetInvalidCount(InvalidComponentHandle handle)
```

Functional Details

Elements that are multiply invalidated are counted once.

Return Value

Returns the number of invalid elements.

See Also

DXCreateInvalidComponentHandle, DXFreeInvalidComponentHandle

13.3, “Invalid Data” on page 133.

DXGetItemSize

Function

Returns the size in bytes of each individual item of an Array.

Syntax

```
#include <dx/dx.h>

int DXGetItemSize(Array a)
```

Functional Details

The routine takes into account the type, category, rank, and shape of Array **a**. (String Arrays are implemented as arrays of type **TYPE_STRING**, **rank 1**, and **shape max_string_length+1**.)

Return Value

Returns the size in bytes or returns 0.

See Also

DXGetCategorySize, DXGetTypeSize

11.3, “Array Class” on page 101.

DXGetMember

Function

Gets a named member of a Group.

Syntax

```
#include <dx/dx.h>

Object DXGetMember(Group g, char *name)
```

Functional Details

The routine returns the member of Group **g** that has been assigned the name **name**. Because the attribute is optional, a name may not have been assigned to a given Group member. In that case, the member can be retrieved by index with **DXGetEnumeratedMember**.

Return Value

Returns the member or returns **NULL** but does not set an error code if the member does not exist.

See Also

DXGetEnumeratedMember, DXGetMemberCount, DXNewGroup, DXSetMember

"Generic Operations" on page 98.

DXGetMemberCount

Function

Retrieves the numerical count of members in a Group.

Syntax

```
#include <dx/dx.h>

Group DXGetMemberCount(Group g, int *n)
```

Functional Details

Group **g** can be generic or a subclass of Group (such as Series, MultiGrid, or Composite Field).

Return Value

Returns a pointer to group **g** and (in ***n**) the number of members in Group **g** or returns **NULL** and sets an error code if **g** is not a Group.

See Also

DXGetEnumeratedMember, DXGetMember, DXNewGroup

"Generic Operations" on page 98.

DXGetMeshArrayInfo

Function

Returns the number of terms and the terms of a Mesh Array.

Syntax

```
#include <dx/dx.h>

MeshArray DXGetMeshArrayInfo(MeshArray a, int *n, Array *terms)
```

Functional Details

If **n** is not **NULL**, the routine returns (in ***n**) the number of terms in the product **a**. If **terms** is not **NULL**, it returns (in ***terms**) the terms of the product.

Mesh Arrays are generally used to specify "connections" components as combinations of lower-dimensional Arrays. In their simplest, form Mesh Arrays are used to combine k Path Arrays into a regular k-dimensional grid, such as regular grids of cubes or quadrilaterals. In order to make this common case simple to use, an alternative, **DXQueryGridConnections** is provided to give quick access to the dimensionality and counts of regular grids.

Array Handles offer a simple mechanism for accessing the individual elements of a Mesh Array without expansion.

Return Value

Returns **a** or returns **NULL** and sets an error code.

See Also

DXCreateArrayHandle, DXGetMeshOffsets, DXMakeGridConnections, DXNewMeshArray, DXNewMeshArrayV, DXQueryGridConnections, DXSetMeshOffsets

“Mesh Arrays” on page 105.

DXGetMeshOffsets

Function

Gets the offset of a partition within the original Field after partitioning.

Syntax

```
#include <dx/dx.h>

MeshArray DXGetMeshOffsets(MeshArray a, int *offsets)
```

Functional Details

The offsets are specified as an Array of integers, one for each dimension of the mesh, specifying the offset along that dimension of the partition in the original Field. In the case where a Mesh Array is used to define a regular grid of connections that is a part of a partitioned Field, it is useful to know the offset of the partition in the original Field.

DXGetMeshOffsets works only on fully regular Mesh Arrays (e.g., those that contain only Path Arrays). Mesh Arrays containing any other type of array will cause an error. In the event that this routine is passed to an Array that is not the result of partitioning, zeros are returned.

Return Value

Returns **a** or returns **NULL** and sets an error code.

See Also

DXGetMeshArrayInfo, DXGetPathOffset, DXNewMeshArray, DXNewMeshArrayV

“Mesh Arrays” on page 105.

DXGetModuleId

Function

Get a unique identifier for each instance of a module.

Syntax

```
#include <dx/dx.h>

Pointer DXGetModuleId()
```

Functional Details

DXGetModuleId returns a pointer to a unique identifier **id**, which can be used to generate unique cache tags (e.g., for use by asynchronous modules).

Return Value

Returns **NULL** or returns **ERROR** and sets an error code.

See Also

DXSetCacheEntry, DXSetCompareModuleId, DXSetCopyModuleId, DXReadyToRun

12.5, “Cache” on page 121.
12.11, “Asynchronous Services” on page 129

DXGetNextHashElement

Function

Returns the next element in a hash table

Syntax

```
#include <dx/dx.h>

Element DXGetNextHashElement(HashTable hashTable)
```

Functional Details

The elements are returned in no predefined order. **DXInitGetNextHashElement** must be called before any calls to **DXGetNextHashElement** are made.

Element is defined as:

```
typedef Pointer Element;
```

Return Value

Returns the next element or returns **NULL** if there are no more elements to return.

See Also

`DXCreateHash, DXInitGetNextHashElement`

13.5, “Hashing” on page 139.

DXGetNextInvalidElementIndex

Function

Returns the index of the next invalid element.

Syntax

```
#include <dx/dx.h>

int DXGetNextInvalidElementIndex(InvalidComponentHandle handle)
```

Functional Details

Returns the index of the next invalid element after the index returned on the prior call (or zero, if this is the first call), given invalid-component handle **`handle`**.

Return Value

Returns the index of the next invalid element or returns -1 if there are no more invalid elements.

See Also

`DXCreateInvalidComponentHandle, DXFreeInvalidComponentHandle`13.3, “Invalid Data” on page 133 .

DXGetNextValidElementIndex

Function

Returns the index of the next valid element.

Syntax

```
#include <dx/dx.h>

int DXGetNextValidElementIndex(InvalidComponentHandle handle)
```

Functional Details

Returns the index of the next valid element after the index returned on the prior call (or zero, if this is the first call), given invalid-component handle **`handle`**.

Return Value

Returns the index of the next valid element or returns -1 if there are no more valid elements.

See Also

DXCreateInvalidComponentHandle, DXFreeInvalidComponentHandle

13.3, "Invalid Data" on page 133.

DXGetObjectClass

Function

Returns the class of an Object.

Syntax

```
#include <dx/dx.h>

Class DXGetObjectClass(Object o)
```

Functional Details

The class of Object **o** is one of the following:

CLASS_ARRAY	**CLASS_INTERPOLATOR**	**CLASS_PRODUCTARRAY**
CLASS_CAMERA	**CLASS_LIGHT**	**CLASS_REGULARARRAY**
CLASS_CLIPPED	**CLASS_MESHARRAY**	**CLASS_SCREEN**
CLASS_COMPOSITEFIELD	**CLASS_MULTIGRID**	**CLASS_SERIES**
CLASS_CONSTANTARRAY	**CLASS_OBJECT**	**CLASS_STRING**
CLASS_FIELD	**CLASS_PATHARRAY**	**CLASS_XFORM**
CLASS_GROUP	**CLASS_PRIVATE**	

All valid Objects have a class that identifies what kind of information the Object contains and what Data Explorer library functions can be called using this Object.

For Objects of **CLASS_GROUP** and **CLASS_ARRAY**, they are further categorized into subclasses. See the **DXGetGroupClass** and **DXGetArrayClass** routines for additional details.

Return Value

Returns the class of an Object. Invalid Objects return **CLASS_MIN**, **CLASS_MAX**, or **CLASS_DELETED**.

See Also

DXGetArrayClass, DXGetGroupClass

"Object Routines" on page 119.

DXGetObjectTag, DXSetObjectTag

Function

Manipulates unique Object identifiers.

Syntax

```
#include <dx/dx.h>

int DXGetObjectTag(Object o)
Object DXSetObjectTag(Object o, int tag)
```

Functional Details

Every Object is assigned a unique nonzero integer tag when it is created. In addition, the executive sets the Object tag of Objects passed to and returned from modules by using **DXSetObjectTag**. This tag is used, for example, by the cache system to identify Objects. **DXGetObjectTag** returns the Object identifier tag for Object **o**. **DXGetObjectTag** can be used with the cache routines **DXGetCacheEntry** and **DXSetCacheEntry** to cache intermediate results between calls to a module.

Note: **DXSetObjectTag** is intended for use only by the executive and should not be used by module writers.

Return Value

Returns the Object identifier or returns 0 and sets an error code.

See Also

DXGetCacheEntry, DXSetCacheEntry

"Object Routines" on page 119.

DXGetPart

Function

Returns the parts of an Object by index.

Syntax

```
#include <dx/dx.h>

Field DXGetPart(Object o, int n)
```

Functional Details

The parts of a Group may be indexed by calling **DXGetPart** with successive values of **n** starting with 0 until **NULL** is returned. This call is equivalent to **DXGetPartClass** with the class parameter set to **CLASS_FIELD**.

Note: The **DXGetPart**, **DXGetPartClass**, and **DXSetPart** routines are useful primarily for prototyping or in cases where their convenience outweighs efficiency concerns. The **DXProcessParts** routine can often be used for the same purposes with better efficiency.

Return Value

Returns the **n**th part or returns **NULL**.

See Also

DXGetPartClass, DXProcessParts, DXSetPart

"Parts" on page 100.

DXGetPartClass

Function

Returns by index only those parts belonging to the specified class.

Syntax

```
#include <dx/dx.h>

Object DXGetPartClass(Object o, int n, Class class)
```

Functional Details

Performs a depth-first traversal of the Object **o** and returns a reference to the **n**th (**n**>= 0) occurrence of a subObject with the requested **class**. The parts of a group may be enumerated by calling **DXGetPartClass** with successive values of **n** starting at 0 until **NULL** is returned.

For applying a function to every Field in a Group, **DXProcessParts** is a more efficient interface.

Return Value

Returns the **n**th subObject of the requested class or returns **NULL**.

See Also

DXGetPart, DXProcessParts, DXSetPart

"Parts" on page 100.

DXGetPathArrayInfo

Function

Returns the number of points referred to in a Path Array.

Syntax

```
#include <dx/dx.h>

PathArray DXGetPathArrayInfo(PathArray a, int *count)
```

Functional Details

If **count** is not **NULL**, this routine returns in ***count** the number of points referred to by the Path Array **a**; this is one more than the number of line segments in **a**.

Path Arrays are used to define the regular constituents of a connections grid. As such, they are most often found as members of Mesh Arrays. While it is possible to use Path Arrays directly to define 1-dimensional regular connections, they will not be recognized through the usual **DXQueryGridConnections** interface, and, in some cases, will not be handled without expansion. It is therefore preferable to use **DXMakeGridConnections** and **DXQueryGridConnections** to define and access 1-dimensional regular grids.

Array handles offer a simple mechanism to access individual elements of a Path Array without expansion.

Return Value

Returns **a** or returns **NULL** and sets an error code.

See Also

DXCreateArrayHandle, DXGetPathOffset, DXMakeGridConnections, DXNewPathArray, DXQueryGridConnections, DXSetPathOffset

“Path Arrays” on page 104.

DXGetPathOffset

Function

Gets the offset of a Path Array in the original Field after partitioning.

Syntax

```
#include <dx/dx.h>

PathArray DXGetPathOffset(PathArray a, int *offset)
```

Functional Details

Gets the offset value **offset** for this portion of the Path Array **a** relative to the original grid. In the case where a Path Array is used to define a regular grid of connections that is a part of a partitioned Field, it is useful to know the offset of the partition within the original Field.

Path Arrays are typically used as constituents in Mesh Arrays that define regular or partially regular connections grids of one or more dimensions. In that case, the mesh offsets of the partition within the original mesh are generally accessed at the Mesh Array level through calls to **DXSetMeshOffsets** and **DXGetMeshOffsets**.

Return Value

Returns **a** or returns **NULL** and sets an error code.

See Also

DXGetMeshOffsets, DXGetPathArrayInfo, DXNewPathArray, DXSetMeshOffsets

“Path Arrays” on page 104.

DXGetPickPoint

Function

Returns the pick point in world coordinates.

Syntax

```
#include <dx/dx.h>

Error DXGetPickPoint(Field picks, int poke, int pick, Point *point)
```

Functional Details

Given **picks** pick information, poke number **poke**, and pick number **pick**, the routine returns the pick point in **point**.

A **Point** is defined as follows:

```
typedef struct point {
    float x, y, z;
} Point, Vector;
```

Return Value

Returns **OK** or returns **NULL** and sets an error code.

See Also

DXQueryPickCount, DXQueryPickPath, DXQueryPokeCount, DXTraversePickPath

13.6, “Pick-Assistance Routines” on page 142.

DXGetPixels

Function

Returns a pointer to the data in an image Field.

Syntax

```
#include <dx/dx.h>

RGBColor *DXGetPixels(Field i)
```

Functional Details

Returns a pointer to the Array of RGBColors of an image **i**. The length of this array can be determined using **DXGetImageSize**.

Return Value

Returns a pointer or returns **NULL** and sets an error code.

See Also

`DXGetImageSize, DXMakeImage`

15.9, “Image Fields” on page 156.

DXGetPrivateData

Function

Returns the private data pointer associated with a private Object.

Syntax

```
#include <dx/dx.h>

Pointer DXGetPrivateData(Private p)
```

Functional Details

This routine is used to access the private data pointer specified when **`p`** was created.

Return Value

Returns the private data pointer.

See Also

`DXNewPrivate`

11.5, “Private Class” on page 106.

DXGetProductArrayInfo

Function

Returns the number of terms and the terms of a Product Array.

Syntax

```
#include <dx/dx.h>

ProductArray DXGetProductArrayInfo(ProductArray a, int *n, Array *terms)
```

Functional Details

If **`n`** is not **`NULL`**, this routine returns in **`*n`** the number of terms in the product **`a`**. If **`terms`** is not **`NULL`**, it returns in **`*terms`** the terms of the product.

Product Arrays provide a compact method for specifying regular and partially regular “positions” components. In their simplest form, a regular n-dimensional grid may be defined by combining n Regular Arrays, each of which specifies a set of points along some n-dimensional delta vector. Partially regular “positions” components may be specified compactly by combining regular and irregular terms.

`DXGetProductArrayInfo` allows access to the constituent terms of the Product Array and is useful in cases where the terms may be handled independently, or when knowledge of the separate terms make it possible to process the Product Array without expansion. Array handles also provide a mechanism to access individual elements of a Product Array without expansion.

Return Value

Returns **`a`** or returns **`NULL`** and sets an error code.

See Also

`DXCreateArrayHandle`, **`DXNewProductArray`**, **`DXNewProductArrayV`**, **`DXQueryGridPositions`**

"Product Arrays" on page 105.

DXGetRegularArrayInfo

Function

Returns the number of items, the origin, and the delta of a Regular Array.

Syntax

```
#include <dx/dx.h>

RegularArray DXGetRegularArrayInfo(RegularArray a, int *count,
                                   Pointer origin, Pointer delta)
```

Functional Details

If **`count`** is not **`NULL`**, this routine returns in **`*count`** the number of points. If **`origin`** is not **`NULL`**, it returns in **`*origin`** the position of the first point. If **`delta`** is not **`NULL`**, it returns in **`*delta`** the spacing between the points. Both **`origin`** and **`delta`** must point to buffers large enough to hold one item of the type of **`a`**. The information about **`a`** may be obtained by calling **`DXGetArrayInfo`**.

Regular Arrays provide a compact representation for a sequence of **`count`** points beginning at **`origin`** and extending in the direction specified by the **`delta`** vector and spaced **`delta`** apart. The dimensionality of **`origin`** and **`delta`** may be found through a call to **`DXGetArrayInfo`**. By accessing the origin and delta information directly, it may be possible to process Regular Arrays without expansion. Array handles provides a mechanism for accessing individual elements of a Regular Array without expansion.

Return Value

Returns **`a`** or returns **`NULL`** and sets an error code.

See Also

`DXCreateArrayHandle`, **`DXGetArrayInfo`**, **`DXNewRegularArray`**

"Regular Arrays" on page 104.

DXGetScreenInfo

Function

Returns information about a Screen Object.

Syntax

```
#include <dx/dx.h>

Screen DXGetScreenInfo(Screen s, Object *o, int *position, int *z)
```

Functional Details

Returns the Object being transformed and the screen transformation parameters from Screen Object **s**. If **o** is not **NULL**, then the Object being transformed by the screen transformation is returned in **o**. Similarly, if position is not **NULL**, then the type of screen transformation is returned in **position**. Finally, if **z** is not **NULL**, the depth of the Screen Object is returned in **z**.

The value returned in **position** will be one of the following:

- **SCREEN_VIEWPORT**—The origin of **o** is in viewport-relative coordinates. **o** will be centered in the viewport when displayed.
- **SCREEN_PIXEL**—The origin of **o** is in pixel coordinates. **o** will be centered about the specified pixel coordinate when displayed.
- **SCREEN_WORLD**—The origin of **o** is in world coordinates. **o** will be located in the world coordinate system.
- **SCREEN_STATIONARY**—**o** is parallel to the screen, but in its own coordinate system.

When position is **SCREEN_STATIONARY**, then **z** will take one of the following values:

< 0	Object **o** is to be displayed behind the other Objects in the scene.
0	Object **o** is to be displayed in the middle of the scene.
> 0	Object **o** is to be displayed in front of the other Objects in the scene.

Return Value

Returns **s** or returns **NULL** and sets an error code if **s** is not a Screen Object.

See Also

DXNewScreen, DXSetScreenObject

15.5, "Screen Class" on page 154.

DXGetSeriesMember

Function

Returns an indexed member from a Series Object.

Syntax

```
#include <dx/dx.h>

Object DXGetSeriesMember(Series s, int n, float *position)
```

Functional Details

Retrieves the Object specified by the zero-based index **n** from the Series **s**. If **s** has an **n**th member, and **position** is not **NULL**, then the value of that Object's position in the Series will be returned in ***position**.

A Series is intended to represent a single Field sampled across some parameter, such as time or temperature. **position** contains the value of this sampled parameter.

Series members cannot be retrieved by the Series **position** value. They must be retrieved by index value **n**.

Return Value

Returns the **n**th member or returns **NULL** if **n** is out of range. Returns **NULL** and sets an error code.

See Also

DXGetEnumeratedMember, DXGetMember, DXGetMemberCount, DXNewSeries, DXSetSeriesMember

"Series Groups" on page 99.

DXGetString

Function

Gets a pointer to the contents of a String Object.

Syntax

```
#include <dx/dx.h>

char *DXGetString(String s)
```

Functional Details

Returns a pointer to the **NULL**-terminated character string contained in a String Object **s**. The contents of the string must not be modified.

Return Value

Returns a pointer to the string value or returns **NULL** and sets an error code.

See Also

DXNewString

11.4, "String Class" on page 106.

DXGetStringAttribute

Function

Retrieves a named attribute from an Object, verifies that it contains a string, and returns a pointer to that string.

Syntax

```
#include <dx/dx.h>

Object DXGetStringAttribute(Object o, char *name, char **x)
```

Functional Details

The attribute to be retrieved from Object **o** is specified by the string **name**. It then verifies that the attribute contains a string value. If **x** is not **NULL**, a pointer to the string is returned in ***x**.

Return Value

Returns **o** returns **NULL** and does not set an error code.

See Also

DXGetAttribute, DXGetEnumeratedAttribute, DXGetFloatAttribute, DXGetIntegerAttribute, DXSetAttribute, DXSetFloatAttribute, DXSetIntegerAttribute, DXSetStringAttribute

“Object Routines” on page 119.

DXGetTime

Function

Returns the elapsed time, in seconds, from system initialization.

Syntax

```
#include <dx/dx.h>

double DXGetTime()
```

Functional Details

The resolution of this measurement is limited by the timing information available from the underlying operating system.

Generally, calls to **DXGetTime** should be used in pairs to measure the amount of time elapsed between two events during a single execution of Data Explorer.

Return Value

Returns the elapsed time.

See Also

`DXMarkTime, DXMarkTimeLocal, DXPrintTimes, DXTraceTime`

12.2, "Timing" on page 116.

DXGetType

Function

Returns the type, category, rank, and shape of an Object.

Syntax

```
#include <dx/dx.h>

Object DXGetType(Object o, Type *t, Category *c, int *rank, int *shape)
```

Functional Details

If **t** is not **NULL**, this routine returns the type of **g** in ***t**. If **c** is not **NULL**, it returns the type of **g** in ***c**. If **rank** is not **NULL**, it returns the rank of **g** in ***rank**. If **shape** is not **NULL**, it returns the shape Array of **g** in ***shape**. **shape** must point to an Array at least ***rank** in length.

The type is one of the following:

```
TYPE_BYTE           TYPE_HYPER        TYPE_SHORT
TYPE_UBYTE          TYPE_INT          TYPE_USHORT
TYPE_DOUBLE         TYPE_UINT         TYPE_STRING
TYPE_FLOAT
```

The category is either **CATEGORY_REAL** or **CATEGORY_COMPLEX**.

Array Objects are always typed. Fields are typed if they contain a “data” component; their type is the same as that of the “data” component. Series, MultiGrids, and Composite Fields are typed if they contain typed Fields. Generic Groups may be typed by explicitly calling **DXSetGroupType**. If typed, all Fields contained in the Group must match the type. Other Objects do not contain type information.

Return Value

Returns **o** only if there is a type associated with **o** or returns **NULL** without setting an error code.

See Also

DXGetArrayInfo, DXSetGroupType, DXUnsetGroupType

“Setting Data Types” on page 120.

DXGetValidCount

Function

Returns the number of valid elements in an invalid-component handle.

Syntax

```
#include <dx/dx.h>

int DXGetValidCount(InvalidComponentHandle handle)
```

Functional Details

Elements that are multiply validated are counted once.

Return Value

Returns the number of valid elements.

See Also

DXCreateInvalidComponentHandle, DXFreeInvalidComponentHandle

13.3, “Invalid Data” on page 133.

DXGetXformInfo

Function

Extracts information from a Transform Object.

Syntax

```
#include <dx/dx.h>

Xform DXGetXformInfo(Xform x, Object *o, Matrix *m)
```

Functional Details

Extracts the Object being transformed and the transformation matrix from a Transform Object **x**. If **o** is not **NULL**, then the Object being transformed by the transformation contained in **t** is returned in **o**. Similarly, if **m** is not **NULL**, then the transformation contained in **t** is returned in **m**.

Return Value

Returns **t** or returns **NULL** and sets an error code if **t** is not a Transform.

See Also

DXNewXform, DXSetXformObject

15.4, “Xform Class” on page 154.

DXGrow, DXGrowV

Function

Add information from neighboring partitions to a Composite Field.

Syntax

```
#include <dx/dx.h>

Object DXGrow(Object object, int n, Pointer fill, ...)
Object DXGrowV(Object object, int n, Pointer fill, char **components)
```

Functional Details

Recursively add boundary information to each partition of any Composite Fields encountered in **object**.

The depth of overlap is defined by **n**. The treatment of the boundary of the Field is specified by the **fill** parameter: specify **GROW_NONE** for no expansion at the exterior boundary of the entire field; **GROW_REPLICATE** to expand at the exterior boundary by replicating the nearest edge values; **GROW_NOFILL** to expand the exterior boundary by leaving space for the extra data but leaving their value undefined; any other value of **fill** must be a pointer to a data item of the correct type to expand at the boundary. It is used as the fill value. For **DXGrowV**, the **components** Array contains a **NULL**-terminated list of components to be grown; all others remain unaffected. For **DXGrow**, the final arguments after **n** consist of a **NULL**-terminated list of the components to be grown. For each component that is grown, the original component is renamed to "original *component*" before the grown component is created.

Return Value

Returns the input Object with the overlapping data added or returns **NULL** and sets an error code. If **o** is a field, returns **o** unmodified.

See Also

DXQueryOriginalMeshExtents, DXQueryOriginalSizes, DXShrink

13.4, "Growing and Shrinking Partitioned Data" on page 137.

DXImportCDF

Function

Imports data from an CDF file.

Syntax

```
#include <dx/dx.h>

Object DXImportCDF(char *filename, char **variable, int *start, int
                   *end, int *delta)
```

Functional Details

Imports data from a CDF (Common Data Format) **filename**. If **variable** is specified, only fields matching the list in **variable** are imported. If **variable** is NULL, all fields are read in and placed in a group. If the CDF contains records, then a series is imported and **start**, **end**, and **delta** can be used to specify which records are imported.

For additional information on the CDF file format, see Appendix B, "Importing Data: File Formats" on page 241 in *IBM Visualization Data Explorer User's Guide*.

Return Value

Returns a pointer to the Object or returns NULL and sets an error code.

See Also

DXImportNetCDF, DXImportDX, DXImportCM, DXImportHDF

DXImportCM

Function

Imports data from an Data Explorer colormap file.

Syntax

```
#include <dx/dx.h>

DXImportCM(char *filename,char **variable)
```

Functional Details

Imports data from a Data Explorer colormap file **filename**. (These files can be saved using **Save As** from the File menu of the Colormap Editor). **variable** can be "colormap" or "opacity". If **variable** is NULL, then both the colormap and the opacity map are imported and placed in a group.

Return Value

Returns a pointer to the Object or returns NULL and sets an error code.

See Also

DXImportNetCDF, DXImportDX, DXImportHDF, DXImportCDF

DXImportDX

Function

Imports data from a Data Explorer file.

Syntax

```
#include <dx/dx.h>

Object DXImportDX(char *filename, char **variable, int *start, int *end,
                  int *delta)
```

Functional Details

Imports data from a Data Explorer file specified by the **filename** parameter. The **variable** parameter specifies a **NULL**-terminated list of strings that identify which variables to import. This parameter identifies Objects in the file that have the names specified by **variable**. If more than one variable is specified, the Objects are collected together and a Group is returned.

For Series data, ***start, *end,** and ***delta** are used to control which indexed Series members are read in. If **start** is **NULL**, it defaults to the beginning of the series; if **end** is **NULL**, it defaults to the end of the series; if **delta** is **NULL**, it defaults to one.

For additional information on the Data Explorer file format, see Appendix B, "Importing Data: File Formats" on page 241. in *IBM Visualization Data Explorer User's Guide*.

Return Value

Returns a pointer to the Object or returns **NULL** and sets an error code.

See Also

DXImportHDF, DXImportNetCDF,DXImportHDF , DXImportCM, DXImportCDF

"Data Explorer Format Files" on page 110.

DXImportHDF

Function

Imports data from an HDF file.

Syntax

```
#include <dx/dx.h>

Field DXImportHDF(char *filename, char *variable)
```

Functional Details

Imports data from an HDF (Hierarchical Data Format) file. **`variable`**, though represented as a string, is a number corresponding to the (zero-based) position of the dataset in the file **`filename`**. If **`variable`** is NULL, all fields are read in and placed in a group.

For additional information on the HDF file format, see Appendix B, "Importing Data: File Formats" on page 241 in *IBM Visualization Data Explorer User's Guide*.

Return Value

Returns a pointer to the Object or returns NULL and sets an error code.

See Also

`DXImportNetCDF, DXImportDX, DXImportCM, DXImportCDF`

DXImportNetCDF

Function

Imports data from a netCDF file.

Syntax

```
#include <dx/dx.h>

Object DXImportNetCDF(char *filename, char **variable, int *start,
                      int *end, int *delta)
```

Functional Details

The routine creates a new Field or Group Object to hold the data that are to be imported. The **`filename`** parameter is the name of a data file in netCDF format. The **`variable`** parameter specifies a **`NULL`**-terminated list of strings that identify the variables to be imported. If more than one is specified, the Objects are collected together and a Group is returned.

For Series data, **`*start`**, **`*end`**, and **`*delta`** are used to control which indexed Series members are read in. If **`start`** is **`NULL`**, it defaults to the beginning of the series; if **`end`** is **`NULL`**, it defaults to the end of the series; if **`delta`** is **`NULL`**, it defaults to 1 (one).

For additional information on the Data Explorer requirements and conventions for the netCDF file, see Appendix B, "Importing Data: File Formats" on page 241 in *IBM Visualization Data Explorer User's Guide*.

Return Value

Returns a pointer to the Object or returns **`NULL`** and sets an error code.

See Also

DXImportDX, DXImportHDF,DXImportHDF , DXImportCM, DXImportCDF

"netCDF Data" on page 111.

DXInitGetNextHashElement

Function

Initializes a pointer for DXGetNextHashElement.

Syntax

```
#include <dx/dx.h>

Error DXInitGetNextHashElement(HashTable hashtable)
```

Functional Details

Generally, a program retrieves entries from a hash table by calling **DXQueryHashElement** to access arbitrary elements, but sometimes it is convenient to access all elements, regardless of their order. In that case, the program calls **DXInitGetNextHashElement** to initiate this indexing by initializing the pointer that **DXGetNextHashElement** uses for iterating **hashtable**. **DXInitGetNextElement** initializes the required pointer to zero (0).

Return Value

Returns **OK** or returns **ERROR** and sets an error code.

See Also

DXCreateHash, DXGetNextHashElement, DXInsertHashElement, DXQueryHashElement

13.5, "Hashing" on page 139.

DXInitModules

Function

Performs necessary initialization when using DXCallModule in a stand-alone program or outboard module.

Syntax

```
#include <dx/dx.h>

void DXInitModules( );
```

Return Value

No return value.

See Also

DXCallModule, DXModSet..., DXSetModule...

DXInitGetNextInvalidElementIndex, DXInitGetNextValidElementIndex

Function

Prepares an invalid-component handle for iteration through the invalid or valid elements.

Syntax

```
#include <dx/dx.h>

Error DXInitGetNextInvalidElementIndex(InvalidComponentHandle handle)
Error DXInitGetNextValidElementIndex(InvalidComponentHandle handle)
```

Functional Details

Prepares the invalid-component handle **handle** for iteration through the contents.

Return Value

Returns **OK** or returns **ERROR** and sets an error code.

See Also

DXCreateInvalidComponentHandle, DXFreeInvalidComponentHandle

13.3, "Invalid Data" on page 133.

DXInsert

Function

Adds a component to a Field.

Syntax

```
#include <dx/dx.h>

Object DXInsert(Object o, Object add, char *name)
```

Functional Details

For each Field in Object **o**, add Object **add** as component **name**. Object **o** can be a single Field, or any Object that can contain Fields (e.g., Groups or Series). If Object **o** is a single Field, Object **add** must be a single Object, usually an Array. If Object **o** is anything else, the Object hierarchy of **o** must match that of **add**, where each Field of **o** matches an Array in **add**. If the Field already contains a **name** component, it is replaced.

Return Value

Returns **o** or returns **NULL** and sets an error code.

See Also

DXExists, DXExtract, DXRemove, DXRename, DXReplace, DXSetComponentValue, DXSwap

11.10, “Component Manipulation” on page 110.

DXInsertHashElement

Function

Inserts an element into a hash table.

Syntax

```
#include <dx/dx.h>

Error DXInsertHashElement(HashTable hashtable, Element element)
```

Functional Details

If a hash function was provided at the time the hash table was created, then that function will be used to derive a pseudokey from the contents of **element**. If no hash function was provided, then the first long integer of **element** is assumed to be the pseudokey.

If there is already an element stored in the hash table with that pseudokey, the behavior will depend on whether or not a compare function was provided at the time the hash table was created. If one was not provided, the already-stored element will be overwritten with the new one. If a compare function was provided, then it will be used to determine whether the keys associated with the elements are in fact unique. If they are, then each will be stored, up to a maximum of 16 for a given pseudokey.

Element is defined as:

```
typedef Pointer Element;
```

Return Value

Returns **OK** or returns **ERROR** and sets an error code.

See Also

DXCreateHash

13.5, “Hashing” on page 139.

DXInterpolate

Function

Interpolates data values in a Field.

Syntax

```
#include <dx/dx.h>

Interpolator DXInterpolate(Interpolator interpolator, int *n,
                           float *points, Pointer result)
```

Functional Details

Interpolates up to ***n** points in the data Object associated with **interpolator**. The **points** parameter is a pointer to a list of sample points to be interpolated. The **result** is a pointer to a buffer large enough to hold ***n** elements of the type of the data Object associated with **interpolator**. The input sample points are interpolated sequentially until a point lying outside the data model is encountered, at which time interpolation terminates. This routine returns in ***n** the number of points that remained to be interpolated when a point outside the data Object is found; this is not considered to be an error.

Points must be of the same dimensionality as the positions in the interpolation Object; thus, (x) points are used to interpolate along the line, (x,y) points are used to interpolate in the plane and (x,y,z) points in 3-dimensional space.

Return Value

This routine returns **interpolator** or returns **NULL** and sets an error code.

See Also

DXLocalizeInterpolator, DXMap, DXMapArray, DXNewInterpolator

13.2, “Interpolation and Mapping” on page 132.

DXInvalidateConnections

Function

Propagates the validity of positions.

Syntax

```
#include <dx/dx.h>

Object DXInvalidateConnections(Object object)
```

Functional Details

Propagates the validity of positions within the Fields of **object** to the connections, faces, or polylines. The validity of the positions is determined from the contents of the “invalid positions” component. A connections, faces, or polylines element will be invalidated if any of its constituent positions are invalid. An “invalid” connections, faces, or polylines component will be created if necessary. If there is no "invalid positions" component, if none of the positions are invalid, or if there is no "connections," "faces," or "polylines," component, there will be no change to **`object`**.

To invalidate positions that become unreferenced because of the action of **`DXInvalidateConnections`**, use **`DXInvalidateUnreferencedPositions`**. To remove invalidated connections, faces, or polylines (and positions), use **`DXCull`**.

Return Value

Returns the updated Object or returns **`NULL`** and sets an error code.

See Also

`DXCull, DXInvalidateUnreferencedPositions`

13.3, "Invalid Data" on page 133.

DXInvalidateDupBoundary

Function

Invalidates all but one of the positions shared between partitions in a Composite Field.

Syntax

```
#include <dx/dx.h>

Object DXInvalidateDupBoundary(Object object)
```

Functional Details

In a Composite Field, positions (and any position-dependent component entries) along a common boundary are shared between partitions. Unless this duplication is taken into account, certain operations (e.g., calculations of statistics or the number of points in a Field) may produce incorrect results. This routine invalidates all but one of the duplicated positions.

Return Value

Returns the input Object with the duplicate positions invalidated or returns NULL and sets an error code.

See Also

`DXCull`

13.3, "Invalid Data" on page 133.

DXInvalidateUnreferencedPositions

Function

Determines which positions in the Fields of the input Object are not referenced by any connections, faces, or polylines element and then invalidates them.

Syntax

```
#include <dx/dx.h>

Object DXInvalidateUnreferencedPositions(Object object)
```

Functional Details

Positions referenced by invalid connection elements (as determined from the contents of an “invalid” connections, faces, or polylines component) are considered to be unreferenced.

Positions are invalidated by use of the “invalid positions” component. If that component does not already exist, one is created.

To remove the invalidated positions, use **DXCull**.

Return Value

Returns the updated Object or returns **NULL** and sets an error code.

See Also

DXCull, DXInvalidateConnections

13.3, “Invalid Data” on page 133.

DXInvertValidity

Function

Inverts the validity state of every element in a specified invalid-component handle.

Syntax

```
#include <dx/dx.h>

Error DXInvertVaildity(InvalidComponentHandle handle)
```

Functional Details

All valid elements in the handle are changed to **DATA_INVALID**; all invalid elements, to **DATA_VALID**.

Return Value

Returns **OK** or returns **ERROR** and sets an error code.

See Also

DXCreateInvalidComponentHandle, DXFreeInvalidComponentHandle

13.3, “Invalid Data” on page 133.

DXIsElementValid, DXIsElementInvalid

Function

Returns the validity of a specified element of an invalid-component handle.

Syntax

```
#include <dx/dx.h>

int DXIsElementValid(InvalidComponentHandle handle, int index)
int DXIsElementInvalid(InvalidComponentHandle handle, int index)
```

Functional Details

The result reflects both the initial conditions of **handle** when **handle** was created using **DXCreateInvalidComponentHandle**, plus any effects of any calls to **DXSetElementInvalid**, **DXSetElementValid**, **DXInvertValidity**, **DXSetAllValid**, and **DXSetAllInvalid**.

Return Value

DXIsElementInvalid returns TRUE (1) if element **index** in the invalid-component handle **handle** is invalid, and returns FALSE (0) otherwise.

DXIsElementValid returns TRUE (1) if element **index** in the invalid-component handle **handle** is valid, and returns FALSE (0) otherwise.

See Also

DXCreateInvalidComponentHandle, DXFreeInvalidComponentHandle

13.3, “Invalid Data” on page 133.

DXIsElementValidSequential, DXIsElementInvalidSequential

Function

Return the validity of a specified element of an invalid-component handle when the queries come in sequential order.

Syntax

```
#include <dx/dx.h>

int DXIsElementValidSequential(InvalidComponentHandle handle, int index)
int DXIsElementInvalidSequential(InvalidComponentHandle handle, int index)
```

Functional Details

Access with either routine is generally faster than with **DXIsElementInvalid** or **DXIsElementValid**.

The result reflects both the initial conditions of **handle** when it was created with **DXCreateInvalidComponentHandle**, and of the effects of any calls to **DXSetElementInvalid**, **DXSetElementValid**, **DXInvertValidity**, **DXSetAllValid**, or **DXSetAllInvalid**.

Note: Accesses must be in sequential order; if they are not, the results may be incorrect.

Return Value

DXIsElementInvalidSequential returns TRUE (1) if element **index** is marked invalid, and returns FALSE (0) otherwise.

DXIsElementValidSequential returns TRUE (1) if element **index** is marked valid, and returns FALSE (0) otherwise.

See Also

DXCreateInvalidComponentHandle, DXFreeInvalidComponentHandle

13.3, “Invalid Data” on page 133.

DXIterateArray

Function

Iterates through an Array.

Syntax

```
#include <dx/dx.h>

Pointer DXIterateArray(ArrayHandle handle, int offset,
                       Pointer last, Pointer scratch)
```

Functional Details

This routine can be used when an Array is accessed sequentially. The Array handle **handle** is that obtained through the use of **DXCreateArrayHandle**.

If the Array is constant, a pointer to the constant value is immediately returned. If the Array is irregular and the given **offset** is 0, the Array data pointer (which is a pointer to the 0th element of the Array) is returned. If the **offset** is not 0, then the **last** parameter should point to the prior Array element, and (**last** + itemSize) is returned, where itemSize is the size in bytes of an item of the Array described by **handle**. If the Array is compact, the desired element is calculated. **scratch** should be of a size large enough to hold a single Array item.

Return Value

Returns a pointer to the next item in an Array or returns **NULL** (but does not set an error) code when there are no more items.

See Also

DXCreateArrayHandle, DXFreeArrayHandle, DXGetArrayEntry, DXGetArrayEntries, DXGetItemSize

“Array Handling” on page 102.

DXLn, DXTri, DXQuad, DXTetra

Function

Constructs a line, triangle, quadrilateral, or tetrahedron from point identifiers.

Syntax

```
#include <dx/dx.h>

Line DXLn(PointId p, PointId q)
Triangle DXTri(PointId p, PointId q, PointId r)
Quadrilateral DXQuad(PointId p, PointId q, PointId r, PointId s)
Tetrahedron DXTetra(PointId p, PointId q, PointId r, PointId s)
```

Functional Details

Each of the connection primitives defined here consists of indices into an Array that stores the coordinate data, which are typically stored in the “positions” component of a Field.

The type definitions for a **Line**, **Triangle**, **Quadrilateral**, and **Tetrahedron** are as follows:

```
typedef struct line {
    PointId p, q;
} Line;

typedef struct triangle {
    PointId p, q, r;
} Triangle;

typedef struct quadrilateral {
    PointId p, q, r, s;
} Quadrilateral;

typedef struct tetrahedron {
    PointId p, q, r, s;
} Tetrahedron;
```

A Point Id is defined as follows:

```
typedef int PointId;
```

Return Value

Returns a line, triangle, quadrilateral, or tetrahedron.

See Also

DXAddArrayData, DXAddLine, DXAddQuad, DXAddTetrahedron, DXAddTriangle, DXNewArray

“Lines, Triangles, Quadrilaterals, Tetrahedra, and Cubes” on page 124.

DXLocalizeInterpolator

Function

Copies an interpolator structure into local memory.

Syntax

```
#include <dx/dx.h>

Interpolator DXLocalizeInterpolator(Interpolator interp)
```

Functional Details

The interpolator structure (**interp**) is copied from shared memory into local memory, on multiprocessor machines with local memory. If the data being interpolated are relatively small and repeatedly accessed, these structures may be copied into local memory for faster access.

The local copy will automatically be freed when the interpolator is deleted.

Return Value

Returns the localized interpolator or returns **NULL** and sets an error code.

See Also

DXInterpolate, DXNewInterpolator

13.2, “Interpolation and Mapping” on page 132.

DXLoopDone

Function

Terminates a loop

Syntax

```
#include <dx/dx.h>

void DXLoopDone(int done)
```

Functional Details

If this routine is called with a value of **done**=1, then any currently executing loop within the macro containing the module calling DXLoopDone will terminate. In this way it acts much like the Done module. Note that if you call DXLoopDone with a value of 1, and then later call it with a value of 0, the loop will still terminate.

Return Value

Does not return a value.

See Also

DXLoopFirst

12.7, “Looping Support” on page 122.

DXLoopFirst

Function

Indicates whether it is the first time through a loop

Syntax

```
##include <dx/dx.h>

int DXLoopFirst()
```

Functional Details

This function is used to determine whether or not it is the first iteration of the loop within the macro containing the module calling DXLoopFirst.

Return Value

Returns 1 if it is the first iteration of a loop. Returns 0 otherwise.

See Also

DXLoopDone

12.7, “Looping Support” on page 122.

DXMakeFloat

Function

Returns a floating-point Array with a single floating-point value.

Syntax

```
#include <dx/dx.h>

Array DXMakeFloat(float f);
```

Functional Details

Creates an Array of type TYPE_FLOAT containing a single specified floating-point value (**f**). The Array can be deleted with **DXDelete()**.

Return Value

Returns the Array or returns **NULL** and sets an error code.

DXMakeGridConnections, DXMakeGridConnectionsV

Function

Construct a grid of regular connections.

Syntax

```
#include <dx/dx.h>

Array DXMakeGridConnections(int n, int count, int count, ...)
Array DXMakeGridConnectionsV(int n, int *counts)
```

Functional Details

Both routines construct an Array of **n**-dimensional regular grid connections (i.e., a set of **n**-dimensional cubes, or hypercubes). For example, a quad mesh is represented by a 2-dimensional grid, while a cube mesh is represented by a 3-dimensional grid.

Note: **DXMakeGridConnections** and **DXMakeGridConnectionsV** set the "element type" for you.

For **DXMakeGridConnections**, the counts are given as the last **n** arguments. The resulting Array is a Mesh Array of **n** terms, where the *k*th term is the Path Array connecting **counts[k]** points; this routine is included to simplify the process of creating the common case of regular grid connections. The number of points along each axis is given by **counts**.

Return Value

Return the Array or return **NULL** and sets an error code.

See Also

DXMakeGridPositions, **DXMakeGridPositionsV**, **DXNewMeshArray**, **DXNewPathArray**, **DXQueryGridConnections**, **DXQueryGridPositions**

“Creating Positions and Connections Grids” on page 103.

DXMakeGridPositions, DXMakeGridPositionsV

Function

Create an n-dimensional grid of regularly spaced positions.

Syntax

```
#include <dx/dx.h>

Array DXMakeGridPositions(int n, int count, int count, int count,...,
            float origin, float origin, ..., float delta, float delta, ...)
Array DXMakeGridPositionsV(int n, int *counts, float *origin, float *deltas)
```

Functional Details

The grids created by these routines are compactly encoded by creating a Product Array of **n** Regular Arrays and are suitable as “positions” components of a Field. This compact encoding is particularly useful when the data lie in or on a lattice of regularly spaced points, since their positions can be computed, thereby saving the space that explicit indexing would require.

DXMakeGridPositions is used to enter the grid’s specifications as individual arguments of the routine’s parameters. **DXMakeGridPositionsV** is used to enter the same arguments as one integer (**n**) and three Arrays.

- The **n** parameter specifies the dimensionality of the grid. The remaining specifications must be consistent with this value.
- The **counts** parameter specifies the number of elements in each of the grid’s dimensions (e.g., 5 and 4 if **n**=2).
- The **origin** parameter specifies the vector that defines the grid’s origin (e.g., 3.2 and 7.1 for a 2-dimensional grid).
- The **delta** parameter specifies the **n**-dimensional vectors that define the delta values to be used (e.g., if **n**=2, this parameter would have four numbers, to specify two 2-vectors).

The grid constructed by either routine is a Product Array of *n* terms, each of which is a Regular Array. The first zero-based term describes a line in an **n**-dimensional space with **counts**[0] points (starting at the origin), each point separated from the previous point on the line by the **n**-dimensional vector described by **deltas**[0] through **deltas**[n-1]. For each of the remaining **n**-1 terms, the *k*th term will have **counts**[k] points (starting at [0 ... 0]), each of which is separated from the previous point on the line by the **n**-dimensional vector described by **deltas**[k*n] through **deltas**[k*n + (n-1)].

The Array created can be deleted with DXDelete. See 2.4, “Memory Management” on page 13.

Return Value

Returns the Array or returns **`NULL`** and sets an error code.

See Also

`DXMakeGridConnections`, **`DXMakeGridConnectionsV`**, **`DXQueryGridConnections`**, **`DXQueryGridPositions`**

"Creating Positions and Connections Grids" on page 103.

DXMakeImage

Function

Creates a new empty image Field.

Syntax

```
#include <dx/dx.h>

Field DXMakeImage(int width, int height)
```

Functional Details

Simplifies creating a Field that represents an image of the specified **`width`** and **`height`**.

An image Field is a regular 2-dimensional grid of "positions" and "connections," with a "colors" component that is floating-point, 3-vector. The sign of the deltas of the "positions" component determines the orientation of the image. This routine creates each of these components, adds them to a Field, and returns the Field.

The Field created can be deleted with DXDelete. See 2.4, "Memory Management" on page 13.

Return Value

Returns the image or returns **`NULL`** and sets an error code.

See Also

`DXGetImageSize`, **`DXGetPixels`**

15.9, "Image Fields" on page 156.

DXMakeInteger

Function

Creates an integer Array with a specified integer value.

Syntax

```
#include <dx/dx.h>

Array DXMakeInteger(int n);
```

Functional Details

Returns an integer Array of TYPE_INT containing a single specified integer value (**n**). The Array can be deleted with **DXDelete()**.

Return Value

Returns the Array or returns **NULL** and sets an error code.

DXMakeString

Function

Returns a String Object.

Syntax

```
#include <dx/dx.h>

String DXMakeString(char *s);
```

Functional Details

Creates a String Object containing a specified string (**s**). The Object can be deleted with **DXDelete()**.

Return Value

Returns the String Object or returns **NULL** and sets an error code.

DXMakeStringList, DXMakeStringListV

Function

Create a string list from a given list of strings.

Syntax

```
#include <dx/dx.h>

Array DXMakeStringList(int n, char *s,...)
Array DXMakeStringListV(int n, char **s)
```

Functional Details

The first form of this routine (**DXMakeStringList)** specifies the strings as the last **n** arguments. The second form (**DXMakeStringListV)** specifies the strings as an Array of strings **s**.

Return Value

Return the string list or return **NULL** and sets an error code.

See Also

DXNewArray

“String List Routines” on page 102.

DXMap

Function

Interpolates data values at sample points.

Syntax

```
#include <dx/dx.h>

Object DXMap(Object object, Object map, char *src, char *dst)
```

Functional Details

This function provides a simple generic tool for interpolation. Object **object** may be either a Field, a Composite Field, or an Array. In the first two cases, the component specified by **src** is used to sample **map**; the results of the interpolation are placed in the component specified by **dst**, and **object** is returned. If **object** is an Array, it is used directly to interpolate **map**, and an Array containing the interpolated values is returned; in this case, the **src** and **dst** parameters are ignored.

If **map** represents a data Field, it must be a Field, a Composite Field, or an Interpolator. The **src** component of this data Field is used to generate the values at the sample points.

If **map** is an Array, this routine creates a resulting Array that consists of the appropriate number of copies of the contents of the **map** Array (which must contain exactly one item). This result is then handled as previously described: if **object** is a Field or a Composite Field, the result Array is added to the Field using the component name specified by **dst**. Otherwise, the resulting Array is returned. This form is used to create a constant data value at all sample points.

DXMapCheck should be called before **DXMap** to check the compatibility of the input **object** and **map**.

Return Value

Returns an Object (see above) or returns **NULL** and sets an error code.

See Also

DXInterpolate, DXMapArray, DXMapCheck, DXNewInterpolator

13.2, “Interpolation and Mapping” on page 132.

DXMapArray

Function

Provides an intermediate-level mapping function.

Syntax

```
#include <dx/dx.h>

Array DXMapArray(Array index, Interpolator map, Array *invalid)
```

Functional Details

This function is lower-level than **DXMap** but higher than **DXInterpolate**.

The parameter **index** specifies an Array containing points to be sampled from the Interpolator **map**. The result is returned as an Array. If **invalid** is not **NULL**, an Array that indicates invalid data in which uninterpolated elements are tagged **DATA_INVALID** is returned. If an invalid-data Array corresponding to **index** exists prior to the call to **DXMapArray**, it should be passed in through ***invalid**. See “DXInterpolate” on page 280 for valid types of **index** and **map** data.

Return Value

Returns **index** or returns **NULL** and sets an error code.

See Also

DXInterpolate, DXMap, DXMapCheck, DXNewInterpolator

13.2, “Interpolation and Mapping” on page 132.

DXMapCheck

Function

Verifies that the types of **input** and **map** are valid to be used as parameters to the **DXMap** routine.

Syntax

```
#include <dx/dx.h>

Object DXMapCheck(Object input, Object map, char *index,
                  Type *type, Category *category, int *rank, int *shape)
```

Functional Details

If **map** is an Array, it must contain a single element. If **map** is not an Array, the type, category, rank, and shape of the input component specified by **index** must match that of the “positions” component of the map. The type, category, rank and shape of the map (and of the data Object produced by this mapping) are returned in the corresponding arguments, **type**, **category**, **rank**, and **shape**.

The type is one of the following:

TYPE_BYTE	**TYPE_HYPER**	**TYPE_SHORT**
TYPE_UBYTE	**TYPE_INT**	**TYPE_USHORT**
TYPE_DOUBLE	**TYPE_UINT**	**TYPE_STRING**
TYPE_FLOAT		

The category is either **CATEGORY_REAL** or **CATEGORY_COMPLEX**.

Return Value

Returns the **input** argument if **input** and **map** are valid for mapping; otherwise, returns **NULL** but does not set an error code.

See Also

DXMap, DXMapArray

13.2, "Interpolation and Mapping" on page 132.

DXMarkTime, DXMarkTimeLocal

Function

Record time marks.

Syntax

```
#include <dx/dx.h>

void DXMarkTime(char *string)
void DXMarkTimeLocal(char *string)
```

Functional Details

DXMarkTime records a "global" time mark relevant to the system as a whole.

DXMarkTimeLocal records a "local" event relevant to one processor (e.g., during a parallel section). This routine is called by the executive at the beginning and end of each module, at the beginning and end of each task or parallel section, and by some system modules.

Both routines store the current time (from **DXGetTime**) and a copy of **string** in a processor local list. The copied version of **string** will be truncated if longer than 16 characters. The number of time marks stored since the last call to **DXPrintTimes** is limited to 2000 events.

Return Value

None.

See Also

DXGetTime, DXPrintTimes, DXTraceTime

12.2, "Timing" on page 116.

DXMessage

Function

Presents a message to the user.

Syntax

```
#include <dx/dx.h>

void DXMessage(char *message, ...)
```

Functional Details

The message string should not contain newline characters, because the **DXMessage** routine formats the message in a manner appropriate to the output medium. For terminal output, this includes prefixing the message with the processor identifier and appending a newline. The **message** may be a **printf** form string, in which case additional arguments may be necessary.

You can also invoke standard error messages by specifying **message** as

```
"#number"
```

where `number` identifies a message in the file `/usr/lpp/dx/messages`. Note that many of the messages require one or more arguments.

Note: In combination with two other library routines, **DXMessage** can be used to form "long" messages. For details, see "DXBeginLongMessage, DXEndLongMessage" on page 201.

Return Value

None.

See Also

DXBeginLongMessage, DXEndLongMessage, DXSetError, DXWarning

12.1, "Error Handling and Messages" on page 114.

DXNeighbors

Function

Returns the neighbors Array of a Field.

Syntax

```
#include <dx/dx.h>

Array DXNeighbors(Field f)
```

Functional Details

For a Field with irregular connections, returns the “neighbors” component of Field **f**. If it does not exist, it is computed and added to the Field before returning. Neighbors are not computed for connections with element type “lines.”

For a Field with regular connections, returns **NULL** without setting the error code because neighbors in a regular grid can be implicitly determined without using additional memory. **DXQueryGridConnections** can be used to determine if the connections are regular or irregular.

The “neighbors” Array is used to indicate which connection elements share faces. For additional details on neighbors, see Chapter 3, “Understanding the Data Model” on page 15 in *IBM Visualization Data Explorer User’s Guide*.

Return Value

Returns the “neighbors” Array or returns **NULL** and sets an error code (unless the Field has regular or “line” connections., in which case no error code is set.

See Also

DXQueryGridConnections

“Standard Components” on page 107.

DXNewAmbientLight

Function

Creates a light Object representing an ambient light source.

Syntax

```
#include <dx/dx.h>

Light DXNewAmbientLight(RGBColor color)
```

Functional Details

Creates a light Object of the color **color** that produces a constant illumination on all rendered Objects regardless of surface orientation.

An **RGBColor** is defined as follows:

```
typedef struct rgbcolor {
   float r, g, b;
} RGBColor;
```

Return Value

Returns the light or returns **NULL** and sets an error code.

See Also

DXNewDistantLight, DXQueryAmbientLight, DXQueryDistantLight

15.8, "Light Class" on page 156.

DXNewArray, DXNewArrayV

Function

Create an irregular Array Object.

Syntax

```
#include <dx/dx.h>

Array DXNewArray(Type t, Category c, int rank, ...)
Array DXNewArrayV(Type t, Category c, int rank, int *shape)
```

Functional Details

The Object created is an Array of Array subclass **CLASS_ARRAY**. Each **item** is a scalar, vector, matrix, or tensor whose rank is specified by **rank** (number of dimensions).

The first form of this routine (**DXNewArray**) specifies the shape as the last **rank** arguments. The second form (**DXNewArrayV**) specifies **shape** as an Array of integers. Each entry in the item is real or complex, as specified by **c**, with coefficients that are integer, single precision, and double precision, according to **t**. The Array initially contains no items.

The type is one of the following:

TYPE_BYTE	**TYPE_HYPER**	**TYPE_SHORT**
TYPE_UBYTE	**TYPE_INT**	**TYPE_USHORT**
TYPE_DOUBLE	**TYPE_UINT**	**TYPE_STRING**
TYPE_FLOAT		

The category is **CATEGORY_REAL or CATEGORY_COMPLEX**.

Return Value

Return the Array or return **NULL** and set an error code.

DXAddArrayData

"Irregular Arrays" on page 101.

DXNewCamera

Function

Creates a new Camera.

Syntax

```
#include <dx/dx.h>

Camera DXNewCamera()
```

Functional Details

Creates a new Camera. A camera defines the position and orientation of the viewer, the volume of interest of the object being viewed, and the size of the image to contain the resulting view.

A summary of how to interpret a camera follows.

The position and orientation of the view are defined by where the viewer is standing, **from**, where the viewer is looking, **to**, and the tilt of the viewer's head, **up**.

The volume of interest of the object being viewed depends on the type of camera. An orthographic camera defines a box that is centered on the **to** point and has an infinite **z** axis lying along the **to-from** vector. The **y** axis is perpendicular to the **to-from** vector in the direction of the **up** vector. The x- and y-dimensions of the box are given by the **width** and **aspect** parameters of the camera, where **aspect** is defined as the ratio of the height to width.

In orthographic projection, objects do not appear smaller as they get more distant, and in fact, distance between the object and the viewer has no effect on the appearance of the object. The distance between the **to** and **from** points is irrelevant; only the direction is important.

The volume of interest defined by a perspective camera is a pyramid with an apex at the **from** position, and a base at the **to** point perpendicular to the **to-from** vector. The width of the base is defined by the angle formed by the sides of the pyramid at the apex and the distance between the **to** and **from** points. The angle formed by the sides of the pyramid is also known as the "field of view" and is the **fov** parameter.

The **fov** is defined as twice the tangent of half the angle (e.g., for a 90-degree sweep, the fov is 2 * tan(45 degrees), or 2.0). The **fov** can also be thought as the ratio of the width of the base to the distance to the viewer (e.g., for a 20-meter wide area from a distance of 10 meters , set the **fov** to 20/10, or 2.0). The base height is defined by the resulting width times the **aspect**.

With a perspective camera, objects appear smaller with increasing distance. When the perspective pyramid is projected onto the image, the sides of the pyramid are made parallel, with a cross-section equal to the base dimensions. This has the effect of widening (in x and y) the objects in front of the pyramid base and compressing (in x and y) the objects behind the pyramid base.

The Camera created can be deleted with DXDelete. See 2.4, “Memory Management” on page 13.

The horizontal size of the image containing the resulting view is defined as the **`resolution`** in pixels, and the vertical size is determined by the **`resolution`** * **`aspect`**. The same aspect ratio is used for both the size of the image and the volume of interest to prevent the object from being stretched in one of the dimensions. The background color of the image is also a parameter to the camera.

By default, the camera is orthographic, looking from the positive z axis toward the origin; x and y each range from *-1* to *+1*. The image is 640 by 480 pixels, with the origin at the center of the image.

Return Value

Returns the Camera or returns **`NULL`** and sets an error code.

See Also

`DXGetCameraMatrix`, **`DXRender`**, **`DXSetBackgroundColor`**, **`DXSetOrthographic`**, **`DXSetPerspective`**, **`DXSetResolution`**, **`DXSetView`**

15.7, “Camera Class” on page 155.

DXNewClipped

Function

Creates a new Clipped Object.

Syntax

```
#include <dx/dx.h>

Clipped DXNewClipped(Object render, Object clipping)
```

Functional Details

Constructs a Clipped Object that instructs the renderer to render the first argument **`render`** clipped by the second argument **`clipping`**. That is, all parts of Object **`render`** that would have been in front of the **`clipping`** Object are removed at render time.

The **`clipping`** Object must have only surface data (no volume data); the colors and opacity of the surface are ignored. Nesting of clipping Objects is not supported, and the clipping Object must be convex. Every volume and translucent surface in a scene must have the same clipping Object.

The Object created can be deleted with DXDelete. See 2.4, “Memory Management” on page 13.

Return Value

Returns the Clipped Object or returns **`NULL`** and sets an error code.

See Also

`DXGetClippedInfo, DXSetClippedObjects`

15.6, "Clipped Class" on page 155.

DXNewCompositeField

Function

Creates a new Composite Field Object.

Syntax

```
#include <dx/dx.h>

CompositeField DXNewCompositeField()
```

Functional Details

A Composite Field Object (a subclass of Group) is a collection of compatible Fields, all having a "positions" component of the same dimensionality, "data" components of the same type, and "connections" of the same "element type." A Composite Field is intended to be interpreted as a collection of Fields that together define a single Field.

Composite Fields are usually created by **`DXPartition`**. They are used to represent the spatial partitioning of one original Field. No member Field may overlap any other, and, where they abut they must share exact vertex locations. If the "connections" component of Composite Fields are regular (i.e., Mesh Arrays), then the "connections" component of each Field must have a "mesh offsets" attribute that indicates where in the original grid the new Field was located.

The type of a Composite Field is set the first time a typed member is added; a typed member is either a Field with a "data" component or a typed Group. All typed members added to a typed Composite Field must match the type of the Composite Field. If a Composite Field has no members, it is untyped.

The Field created can be deleted with DXDelete. See 2.4, "Memory Management" on page 13.

Return Value

Returns the Composite Field or returns **`NULL`** and sets an error code.

See Also

`DXGrow, DXNewGroup, DXNewMultiGrid, DXNewSeries, DXSetMeshOffests`

"Composite Fields" on page 100.

DXNewConstantArray, DXNewConstantArrayV

Function

Create an Array containing constant data.

Syntax

```
#include <dx/dx.h>

ConstantArray DXNewConstantArray (int items, Pointer data, Type t,
                                  Category c, int rank, ...)
ConstantArray DXNewConstantArrayV (int items, Pointer data, Type t,
                                  Category c, int rank, int *shape)
```

Functional Details

Constant Arrays provide a compact mechanism for representing constant values in Array form. All parameters are required:

- **items** specifies the number of items in the Array.
- **data** should point to a memory location containing the constant value to be stored in the Array.
- **t** specifies the type of the data:

TYPE_BYTE	**TYPE_HYPER**	**TYPE_SHORT**
TYPE_UBYTE	**TYPE_INT**	**TYPE_USHORT**
TYPE_DOUBLE	**TYPE_UINT**	**TYPE_STRING**
TYPE_FLOAT		

- **c** specifies the category of the data: **CATEGORY_REAL** or **CATEGORY_COMPLEX**.
- **rank** specifies the rank of the data.
- **shape** (or the remaining **rank** arguments given in the **DXNewConstantArray** form) specify the shape of the data.

The Array created can be deleted with DXDelete. See 2.4, “Memory Management” on page 13.

Return Value

Return the Constant Array or returns **NULL** and sets an error code.

See Also

DXNewArray

“Constant Arrays” on page 105.

DXNewDistantLight

Function

Creates a distant light Object of specified **color** and **direction**.

Syntax

```
#include <dx/dx.h>

Light DXNewDistantLight(Vector direction, RGBColor color)
```

Functional Details

The light source is located at an infinite distance from the scene in the specified direction. Shading from distant lights differs with orientation in relation to the light and not with the distance from the light.

The Light created can be deleted with DXDelete. See 2.4, "Memory Management" on page 13.

A **Vector** is defined as follows:

```
typedef struct point {
    float x, y, z;
} Point, Vector;
```

An **RGBColor** is defined as follows:

```
typedef struct rgbcolor {
   float r, g, b;
} RGBColor;
```

Return Value

Returns the Light or returns **NULL** and sets an error code.

See Also

DXQueryDistantLight

15.8, "Light Class" on page 156.

DXNewField

Function

Creates a new Field Object.

Syntax

```
#include <dx/dx.h>

Field DXNewField()
```

Functional Details

The Field Object is the fundamental Object in Data Explorer. It consists of zero or more named components, usually Arrays, that are accessed with **DXGetComponentValue** or **DXGetEnumeratedComponentValue**. Initially, the Field has no components and is said to be empty; **DXEmptyField** returns "1" for such Fields. Components are inserted in the Field using the function **DXSetComponentValue**.

There are several predefined component names in Data Explorer. The component "positions" generally refers to the points in the data space where the field is sampled; "connections" are the relationships between the "positions" and their interpolation; "data" refers to the values either at the "positions" or for each whole "connections" element.

The Field created can be deleted with DXDelete. See 2.4, "Memory Management" on page 13.

See Chapter 3, "Understanding the Data Model" on page 15 in *IBM Visualization Data Explorer User's Guide* for additional information on the Data Explorer data model.

Return Value

Returns the Field or returns **`NULL`** and sets an error code.

See Also

`DXEmptyField`, **`DXGetComponentValue`**, **`DXGetEnumeratedComponentValue`**, **`DXSetComponentValue`**

11.1, "Field Class" on page 97.

DXNewGroup

Function

Creates a new generic Group Object.

Syntax

```
#include <dx/dx.h>

Group DXNewGroup()
```

Functional Details

A generic Group consists of a number of members that may optionally be named. There is no restriction on the class of Objects in a generic Group and no requirement that the type of the members be the same. This is in contrast to the Composite Field, MultiGrid, and Series subgroups of the class Group, which generally contain members of class Field and are typed. Generic Groups may be explicitly typed using **`DXSetGroupType`** if all the members are of the same type.

Named members are added to a Group with **`DXSetMember`**. Unnamed members are added to a Group with **`DXSetEnumeratedMember`**.

The Group created can be deleted with DXDelete. See 2.4, "Memory Management" on page 13.

Return Value

Returns the Group or returns **NULL** and sets an error code.

See Also

DXNewCompositeField, DXNewMultiGrid, DXNewSeries, DXSetEnumeratedMember, DXSetGroupType, DXSetMember

“Generic Operations” on page 98.

DXNewInterpolator

Function

Creates an Interpolator Object for interpolating in Object **o**.

Syntax

```
#include <dx/dx.h>

Interpolator DXNewInterpolator(Object o, enum interp_init init, float fuzz)
```

Functional Details

Object **o** should be a Field or an Object that contains Fields (e.g., a Group). An Interpolator builds and stores the information about the Field or Fields to speed up searching for and computing data values corresponding to any location enclosed by the positions of the Object. This is particularly useful with irregular meshes.

The initialization type is specified by setting the **init** argument to **INTERP_INIT_DELAY**, **INTERP_INIT_IMMEDIATE** or **INTERP_INIT_PARALLEL**. **INTERP_INIT_DELAY** does not initialize the Interpolator until the information is actually needed. This is fastest if a small number of points will be interpolated. **INTERP_INIT_IMMEDIATE** does all initialization before returning. **INTERP_INIT_PARALLEL** does all initialization in parallel if running in a multiprocessor machine before returning.

The **fuzz** value assigns a fuzz factor to the interpolation process: any sample falling within this distance of a valid primitive of the Object **o** is assumed to be inside that primitive. When this point lies geometrically outside the primitive, an appropriate result is extrapolated. Any positive or zero value is used as the fuzz factor; a negative value indicates that the interpolator should determine its own fuzz factor.

Interpolators can be used with the **DXMap** call. See “DXMap” on page 293 for additional details.

The interpolator created can be deleted with DXDelete. See 2.4, “Memory Management” on page 13.

Return Value

Returns the Interpolator or returns **NULL** and sets an error code.

See Also

DXInterpolate, DXMap, DXMapArray

13.2, “Interpolation and Mapping” on page 132.

DXNewMeshArray, DXNewMeshArrayV

Function

Create an Array that is the product of a set of regular or irregular connection Arrays.

Syntax

```
#include <dx/dx.h>

MeshArray DXNewMeshArray(int n, ...)
MeshArray DXNewMeshArrayV(int n, Array *terms)
```

Functional Details

The terms of the product are given by the Array pointer **terms** (for **DXNewMeshArrayV**) or by the last **n** arguments (for **DXNewMeshArray**).

Mesh Arrays are generally used to specify regular or partially regular “connections” components in compact form as combinations of lower-degree Arrays. For example, a fully regular cubic “connections” component can be created by combining three Path Arrays in a Mesh Array. A fully regular “connections” component is more easily created with **DXMakeGridConnections**.

The Array created can be deleted with DXDelete. See 2.4, “Memory Management” on page 13.

Return Value

Return the Mesh Array or returns **NULL** and sets an error code.

See Also

DXGetMeshArrayInfo, DXGetMeshOffsets, DXMakeGridConnections, DXSetMeshOffsets

“Mesh Arrays” on page 105.

DXNewMultiGrid

Function

Creates a new MultiGrid Object.

Syntax

```
#include <dx/dx.h>
MultiGrid DXNewMultiGrid()
```

Functional Details

The new Object is a Group Object that can contain a Field or Composite Field Objects.

The Fields in a MultiGrid Object must all be the same data type and have the same connections element type, but the combined positions are not required to fill space exactly. The Fields may overlap or there may be space between them. An “invalid positions” component may be used to select the preferred Field in the overlapping case.

The Group created can be deleted with DXDelete. See 2.4, “Memory Management” on page 13.

Return Value

Returns the MultiGrid Object or returns **NULL** and sets an error code.

See Also

DXCreateInvalidArrayHandle, DXNewCompositeField, DXSetMember

“MultiGrid Groups” on page 99.

DXNewPathArray

Function

Creates an Array specifying the linear connections between points.

Syntax

```
#include <dx/dx.h>

PathArray DXNewPathArray(int count)
```

Functional Details

The type of **a** is implicitly set to integer, the rank to 1, and the shape to 2. The **count** parameter specifies the number of positions connected in the path defined by the Path Array.

Path Arrays are used to specify 1-dimensional regular axes of a regular or partially regular “connections” component representing **count**-1 segments along the axis. These are generally combined in Mesh Arrays to create 1-dimensional or higher regular-connection grids.

In the case of fully regular connections (e.g., when one or more path arrays are to be combined in a Mesh Array), it is often easier to use **DXMakeGridConnections**.

The Array created can be deleted with DXDelete. See 2.4, “Memory Management” on page 13.

Return Value

Returns the Path Array or returns **`NULL`** and sets an error code.

See Also

`DXGetPathArrayInfo, DXGetPathOffset, DXMakeGridConnections, DXSetPathOffset`

“Path Arrays” on page 104.

DXNewPrivate

Function

Creates an Object that contains a pointer to private data.

Syntax

```
#include <dx/dx.h>

Private DXNewPrivate(Pointer data, Error (*deleteFunction)(Pointer))
```

Functional Details

The user is responsible for the private data. If the data Object is larger than the single pointer, you also need to specify **`deleteFunction`**, which takes an argument of type **`Pointer`**. If **`deleteFunction`** is not **`NULL`**, it will be called when the Private Object is deleted and it will be passed the pointer **`data`**. In most cases, **`data`** should be a pointer to global memory.

Private objects are useful for storing arbitrary structures in the cache for later use.

Note: Private Objects cannot be used between different nodes when running in distributed mode.

The Object created can be deleted with DXDelete. See 2.4, “Memory Management” on page 13.

Return Value

Returns the Private Object or returns **`NULL`** and sets an error code.

See Also

`DXAllocate, DXGetPrivateData, DXSetCacheEntry`

11.5, “Private Class” on page 106.

DXNewProductArray, DXNewProductArrayV

Function

Create an Array that is the product of a set of regular or irregular position Arrays.

Syntax

```
#include <dx/dx.h>

ProductArray DXNewProductArray(int n, ...)
ProductArray DXNewProductArrayV(int n, Array *terms)
```

Functional Details

All of the Array types must be floating-point and of the same rank and shape. The Array created will have the same rank and shape as the input Arrays. The terms of the product are given by the Array pointer **terms** (for **DXNewProductArrayV**) or by the last **n** arguments (for **DXNewProductArray**).

Product Arrays are most often used to construct regular or partially regular "positions" components. In the case of fully regular positions (e.g., when each of the items is a Regular Array), it is often be easier to use **DXMakeGridPositions**.

The Array created can be deleted with DXDelete. See 2.4, "Memory Management" on page 13.

Return Value

Return the Product Array or returns **NULL** and set an error code.

See Also

DXGetProductArrayInfo, DXMakeGridPositions

"Product Arrays" on page 105.

DXNewRegularArray

Function

Creates an Array containing evenly spaced data.

Syntax

```
#include <dx/dx.h>

RegularArray DXNewRegularArray(Type t, int dim, int n,
                               Pointer origin, Pointer delta)
```

Functional Details

The new Array Object represents a Regular Array of **n** points starting at **origin** with a spacing of **delta**. The rank is assumed to be 1, and the shape is **dim**. Both **origin** and **delta** are assumed to point to items of the same type as the items in **a**.

Type **t** is one of the following:

TYPE_BYTE	TYPE_HYPER	TYPE_SHORT
TYPE_UBYTE	TYPE_INT	TYPE_USHORT
TYPE_DOUBLE	TYPE_UINT	TYPE_STRING
TYPE_FLOAT		

Regular Arrays are most often used as constituents of Product Arrays for the compact representation of regular grids of positions.

In previous versions of the Data Explorer software, regular arrays with zero delta values were used to store constant data. This type of data can now be stored in a Constant Array.

The Array created can be deleted with DXDelete. See 2.4, “Memory Management” on page 13.

Return Value

Returns the Regular Array or returns **`NULL`** and sets an error code.

See Also

`DXCreateArrayHandle, DXGetRegularArrayInfo, DXNewConstantArray`

“Regular Arrays” on page 104.

DXNewScreen

Function

Creates a new Object aligned to the final screen.

Syntax

```
#include <dx/dx.h>

Screen DXNewScreen(Object o, int position, int z)
```

Functional Details

A Screen Object is an Object that maintains a size and alignment with the screen (output image) independent of the camera view and scaling transformations applied to it.

The **position** parameter specifies one of the three options for positioning of the Screen Object as explained in the following material; it must be one of **SCREEN_WORLD**, **SCREEN_PIXEL** or **SCREEN_VIEWPORT**. The **z** parameter determines the relative depth of the Object, as described in the following text.

Three options are provided for the interpretation of translations applied to a Screen Object. First, a translation applied to the Screen Object may specify a new position for the origin of the Screen Object in world space (**SCREEN_WORLD**). Second, a translation applied to the Screen Object may specify a new location for the Screen Object in the image, measured in pixels, where (0,0) refers to the lower-left corner of the image (**SCREEN_PIXEL**). Third, a translation applied to the Screen Object may specify a new location for the Screen Object in the image, measured in viewport-relative coordinates, where (0,0) refers to the lower-left corner of the image and (1,1) refers to the upper-right corner of the image (**SCREEN_VIEWPORT**).

The **z** parameter controls where the Screen Object is displayed relative to all other Objects in the scene. *-1* displays behind, 0 is in the scene, and *+1* is in front of all other Objects.

The Object created can be deleted with DXDelete. See 2.4, “Memory Management” on page 13.

Return Value

Returns the Screen Object or returns **NULL** and sets an error code.

See Also

DXGetScreenInfo, DXSetScreenObject

15.5, “Screen Class” on page 154.

DXNewSeries

Function

Creates a new Series Object.

Syntax

```
#include <dx/dx.h>

Series DXNewSeries()
```

Functional Details

A Series is intended to represent a single field sampled across some parameter, such as time or temperature (e.g., a simulation of a CMOS device across a temperature range). Members of a Series have a position. A copy of the position is found in the “series position” attribute.

For Realization modules, a Series should be a collection of compatible Fields, each with the same “data,” “position,” and “connection” types, and the Series positions of all members should be increasing only or decreasing only for the whole Series.

The type of a Series is set the first time a typed member is added; a typed member is either a Field with a “data” component or a typed Group. All typed members added to a typed Series must match the type of the Series. If a Series has no members, it is untyped.

The Object created can be deleted with DXDelete. See 2.4, “Memory Management” on page 13.

Return Value

Returns the Series or returns **NULL** and sets an error code.

See Also

DXNewCompositeField, DXNewGroup, DXNewMultiGrid, DXSetSeriesMember

“Series Groups” on page 99.

DXNewString

Function

Creates a new String Object.

Syntax

```
#include <dx/dx.h>

String DXNewString(char *s)
```

Functional Details

The new String Object is initialized with a copy of the specified **NULL**-terminated string **s**.

The Object created can be deleted with DXDelete. See 2.4, "Memory Management" on page 13.

Return Value

Returns the Object or returns **NULL** and sets an error code.

See Also

DXGetString

11.4, "String Class" on page 106.

DXNewXform

Function

Creates a new Transform Object.

Syntax

```
#include <dx/dx.h>

Xform DXNewXform(Object o, Matrix m)
```

Functional Details

The new Transform Object represents Object **o**, to which a modeling transform Matrix **m** is to be applied.

The modeling transformation is specified as a **Matrix** that is defined as follows:

```
typedef struct
{
    float A[3][3]
    float b[3];
} Matrix;
```

This definition of a Matrix is sufficient for specifying all 3-dimensional affine transformations (e.g., xA + b, where A is a 3x3 rotation Matrix and b is a 3-dimensional translation vector).

Transforms may be applied hierarchically.

Note: The transformation is not actually applied when the Transform Object is created.

The Object created can be deleted with DXDelete. See 2.4, "Memory Management" on page 13.

Return Value

Returns the Transform Object or returns **NULL** and sets an error code.

See Also

DXApplyTransform, DXGetXformInfo, DXRotateX, DXRotateY, DXRotateZ, DXScale, DXSetXformObject, DXTranslate

15.4, “Xform Class” on page 154.

DXOutputRGB

Function

Writes an image to a file in RGB format.

Syntax

```
#include <dx/dx.h>

Field DXOutputRGB(Field i, int fd)
```

Functional Details

The routine writes image Field **`i`** to file descriptor **`fd`** as three bytes (red, green, and blue) per pixel.

The values of the floating-point "colors" component (ranging from 0.0 to 1.0) are converted into byte values from 0 to 255. Values below 0 and above 1 are changed to 0 and 255 respectively.

The image is written at the current byte offset in the file. **`lseek()`** should be called first to reposition the current read/write pointer to the desired location.

Return Value

Returns **`i`** or returns **`NULL`** and sets an error code.

See Also

15.9, "Image Fields" on page 156.

DXPartition

Function

Divides a Field into partitions.

Syntax

```
#include <dx/dx.h>

Group DXPartition(Field f, int n, int size)
```

Functional Details

Divides **f** into a maximum of **n** spatially local pieces with at least **size** points in each piece.

The Group created can be deleted with DXDelete. See 2.4, “Memory Management” on page 13.

Return Value

Returns **f** (if it is already partitioned) or returns a new Composite Field; otherwise, it returns **NULL** and sets an error code.

See Also

DXGrow, DXShrink

13.1, “Data Partitioning” on page 132.

DXPrint, DXPrintV

Function

Print an Object according to specified formatting options.

Syntax

```
#include <dx/dx.h>

Error DXPrint(Object o, char *options, ...)
Error DXPrintV(Object o, char *options, char **components)
```

Functional Details

In the case of **DXPrint**, the arguments following **options** should consist of a **NULL**-terminated list of strings. These strings specify the components of a Field to which the options should be applied. Alternatively, the components may all be specified together in a **NULL**-terminated Array of strings and supplied as the **components** argument to **DXPrintV**.

If **components** is not specified or is **NULL**, the formatting options are applied to all components.

If **o** is a Group or other Object capable of containing another Object, then it is traversed in a depth-first order, and the appropriate printing options are applied for the particular Object currently being visited. If the Object being printed is not a Field, then the specified component names are ignored.

The formatting options are:

- r - Recursively traverse the Object.
- o - Print only the top level of the Object.
- d - Print first and last 25 items in Arrays, as well as headers.
- D - Print all the items in Arrays as well as headers.
- x - Print in expanded form rather than compact form.
- *n* - Print Object to *n* levels.

See also "Print" on page 244 in *IBM Visualization Data Explorer User's Reference*.

Return Value

Returns **OK** or returns **NULL** and sets an error code.

See Also

DXCategorySize, DXGetType, DXTypeSize

"Setting Data Types" on page 120.

DXPrintAlloc

Function

Prints a summary of memory use.

Syntax

```
#include <dx/dx.h>

void DXPrintAlloc(int t)
```

Functional Details

This routine can be run from Data Explorer, with the Usage module, in script mode or in the user interface. When it is run, memory areas should be quiescent (on a multiprocessor system, for example, no other tasks should be running). The parameter **t** specifies the level of detail of the printout:

0 Prints out a summary of the current use of memory, both in small and in large areas. A typical printout might look like:

```
0: small: 4194304 = hdr 16472 + used 486864 + free 3920
                            + pool 3687048 (limit 4194304)

0: large: 2097152 = hdr 16472 + used 494656 + free 29704
                            + pool 1558120 (limit 54525952)
```

where:

small is the total number of bytes currently managed by the memory manager for the small arena.

large is the total number of bytes currently managed by the memory manager for the large arena.

hdr	is the amount of memory space used by internal data structures.
used	is the amount of memory space allocated for use.
free	is the amount of memory previously used and available for reuse.
pool	is the amount of memory space allocated to Data Explorer but not yet used.
limit	is the largest amount of memory that can be managed by the memory manager.

1 Prints one line for every individual *allocated* block, stating its size and address.

2 In addition to the information for allocated blocks (how = 1), prints the same information for every *freed* block.

Return Value

None.

See Also

`DXAllocate, DXFree`

12.3, "Memory Allocation" on page 116.

DXPrintTimes

Function

Prints time marks.

Syntax

```
#include <dx/dx.h>

void DXPrintTimes()
```

Functional Details

Prints a merged summary of the "global" time events recorded by **`DXMarkTime`** and the "local" time events recorded by **`DXMarkTimeLocal`** since the last call to **`DXPrintTimes`**.

For each global event, the following are printed:

- The identifying tag that was specified by the **`DXMarkTime`** call
- The time of the last previous global event
- The difference in time between that previous event and this event
- The time of this event.

For each local event, the following are printed:

- The processor on which the event occurred
- The time of the last previous local event on this processor or the last previous global event, whichever occurred later
- The difference in time between that previous event and this event

- The time of this event.

All times are printed in seconds. In addition, on some architectures the difference in user and system times are printed, as recorded by the operating system. These times are not printed on the IBM POWER Visualization System, and appear as zeros.

Return Value

None.

See Also

`DXGetTime, DXMarkTime, DXMarkTimeLocal, DXTraceTime`

12.2, "Timing" on page 116.

DXProcessorId

Function

Returns the identifier of the current processor in a multiprocessor machine.

Syntax

```
#include <dx/dx.h>

int DXProcessorId()
```

Functional Details

The processor identifier is a number ranging from *0* to *n-1*, where *n* is the number of processors in use. This routine is not generally needed by modules.

Return Value

Returns the processor identifier. Always returns 0 (zero) for a single processor machine.

See Also

`DXProcessors`

12.8, "Parallelism" on page 123.

DXProcessors

Function

Returns the number of processors.

Syntax

```
#include <dx/dx.h>

int DXProcessors(int n)
```

Functional Details

Queries the number of processors available for running tasks, if **n=0**, and may be used to determine task allocation for parallelism.

Values of **n!=0** are ignored and reserved for future use.

Return Value

Returns the number of processors.

See Also

DXAddTask, DXCreateTaskGroup, DXExecuteTaskGroup, DXProcessorId

12.8, “Parallelism” on page 123.

DXProcessParts

Function

Applies a function to every constituent Field (part) of a specified Object.

Syntax

```
#include <dx/dx.h>

Object DXProcessParts(Object object, Field (*process)(Field, Pointer, int),
                      Pointer args, int size, int copy, int preserve)
```

Functional Details

If the input **Object** is a Field, this routine returns the result of the **process** function on that Field.

If the input **Object** is a Group and **copy** is 1, this routine recursively makes a copy of the Group and all subgroups. In this case, the order of the Fields in the Groups is preserved if **preserve** is 1. If this is not required, set **preserve** to 0 and a more efficient traversal algorithm will be used.

If the input **Object** is a Group and **copy** is 0, it operates directly on the Groups of the input **object**.

In either case, for every Field **f** that is a member of a Group, it makes a call of the form **process(f**, **args**, **size)** and places the result of that call in the output in place of **`f`**. The **`process`** function is intended to return a Field which is the processed version of input Field **`f`**.

Regardless of the value of the **`copy`** parameter, the Field passed to the **`process`** function is the Field from the original Object and not a copy.

The **`size`** parameter specifies the size of the block pointed to by **`args`**. If **`size`** is nonzero, it makes a copy of the argument block and places it in global memory before passing it to **`process`**. The argument must be in global memory because **`DXProcessParts`** may run in parallel; however, if the pointer passed is, for example, just a pointer to an Object that is already in global memory, then **`size`** can be given as 0. **`args`** should not contain pointers to local memory.

If the **`process`** function returns **`NULL`**, and **`preserve`** is 1 or the Field was part of a Series Group or was the entire input Object, the **`NULL`** return value is replaced with an empty Field.

Return Value

Returns the Object, a copy of the Object, or a processed version of it, depending on the parameters; otherwise, it returns **`NULL`** and sets an error code.

See Also

`DXGetPart, DXGetPartClass, DXSetPart`

“Parts” on page 100.

DXPt, DXVec

Function

Constructs a Point or a Vector with the specified coordinates.

Syntax

```
#include <dx/dx.h>

Point DXPt(double x, double y, double z)
Point DXVec(double x, double y, double z)
```

Functional Details

A **Point** or **Vector** is defined as

```
typedef struct point {
    float x, y, z;
} Point, Vector;
```

Return Value

Return the point or vector.

See Also

"Points and Vectors" on page 124.

DXQueryAmbientLight

Function

Returns the color of an Ambient Light.

Syntax

```
#include <dx/dx.h>

Light DXQueryAmbientLight(Light l, RGBColor *color)
```

Functional Details

Determines whether **l** is an Ambient Light and returns in ***color** the information specified when the Light was created.

An **RGBColor** is defined as follows:

```
typedef struct rgbcolor {
   float r, g, b;
} RGBColor;
```

Return Value

Returns **l** (if it is an Ambient Light) or **NULL** (if it is another type) without setting an error code; otherwise, returns **NULL** and sets an error code.

See Also

DXNewAmbientLight

15.8, “Light Class” on page 156.

DXQueryArrayCommon, DXQueryArrayCommonV

Function

Return a type, category, rank, and shape to which all of the Arrays can be converted.

Syntax

```
#include <dx/dx.h>

Error DXQueryArrayCommon(Type *type, Category *category, int *rank, int *shape
                         int n, Array a, ...)
Error DXQueryArrayCommonV(Type *type, Category *category, int *rank, int *shape
                         int n, Array *alist)
```

Functional Details

All Arrays **a** or **alist** are converted to a common type, category, rank, and shape if a common conversion exists. Both routines set ***type**, ***category**, ***rank**, and ***shape** to the new converted values if the pointers to these parameters are not **NULL**.

Arrays with items that have one or more dimensions of shape = **1** are reduced in that dimension to a smaller rank. For example, an array of **1 × n** matrices, rank = **2**, shape = [**1,n**], is reduced to an Array of vectors rank = **1**, shape = [**n**]. This rank and shape are then used for comparison.

Note: This reduction is always performed. The resulting rank may be less than the rank of all Arrays given. (A single **1 × n** matrix results in a rank of **1**.) Table 6 and Table 7 summarize the conversions allowed between types and between categories.

The resulting rank and shape are the reduced versions created as previously described. The resulting type is the simplest type that is sufficient for all Arrays. The resulting category is the simplest category that is sufficient for all Arrays.

The **shape** Array must be preallocated with sufficient memory to store the returned **rank** integers. This will not exceed the maximum rank of the given arrays.

Table 6. Summary of Type Conversions

	Byte	**Unsigned Byte**	**Short**	**Unsigned Short**	**Int**	**Unsigned Int**	**Float**	**Double**
Byte	A	CNS	A	CNS	A	CNS	A	A
Unsigned Byte	CNS	A	A	A	A	A	A	A
Short	CNS	CNS	A	CNS	A	CNS	A	A
Unsigned Short	CNS	CNS	CNS	A	A	A	A	A
Int	CNS	CNS	CNS	CNS	A	CNS	A	A
Unsigned Int	CNS	CNS	CNS	CNS	CNS	A	A	A
Float	CNS	CNS	CNS	CNS	CNS	CNS	A	A
Double	CNS	CNS	CNS	CNS	CNS	CNS	CNS	A

Notes:

CNS = Conversion not supported
A = ANSI 'C' type conversion-semantics

Table 7. Summary of Category Conversions

	Real	**Complex**
Real	Conversion	Conversion
Complex	CNS	Conversion

Notes:

CNS = Conversion not supported
Real→Complex: a → a + 0i

Return Value

Return **OK** and sets the non-**NULL** parameters; otherwise, returns **ERROR** and sets an error code.

See Also

DXArrayConvert, **DXArrayConvertV**, **DXExtractFloat**, **DXExtractInteger**, **DXExtractNthString**, **DXExtractParameter**, **DXExtractString**, **DXQueryArrayConvert**, **DXQueryArrayConvertV**, **DXQueryParameter**

11.8, "Extracting Module Parameters" on page 108.

DXQueryArrayConvert, DXQueryArrayConvertV

Function

Determine if the given Array can be converted to an Array with the given type, category, rank, and shape.

Syntax

```
#include <dx/dx.h>

Error DXQueryArrayConvert(Array a, Type type, Category category, int rank, ...)
Error DXQueryArrayConvertV(Array a, Type type, Category category, int rank, int *shape)
```

Functional Details

The Array can be converted only if **type**, **category**, **rank**, and **shape** are all compatible with **a**.

rank and **shape** are compatible if the rank and shape of **a** and the given **rank** and **shape** differ only by dimensions that have a shape of **1**. For example, an Array of **1** × **n** matrices is compatible with an Array of vectors.

Table 6 on page 324 and Table 7 on page 324 summarize the conversions allowed between types and categories.

Return Value

Return **OK** or returns **NULL** and set an error code.

See Also

DXArrayConvert, **DXArrayConvertV**, **DXExtractFloat**, **DXExtractInteger**, **DXExtractNthString**, **DXExtractParameter**, **DXExtractString**, **DXQueryParameter**

11.8, "Extracting Module Parameters" on page 108.

DXQueryConstantArray

Function

Determines whether an Array contains constant data, and, if so, returns both the number of items and in the array and the data value.

Syntax

```
#include <dx/dx.h>

Array DXQueryConstantArray (Array a, int *num, Pointer data)
```

Functional Details

DXQueryConstantArray determines if Array **a** contains constant data. If so, if **num** is not **NULL**, the number of items contained in **a** is returned in ***num**. If **data** is not **NULL**, the constant value contained in **a** will be copied to the block of memory pointed to by **data**. In this case **data** should point to a block of memory large enough to hold one element of **a**; this size can be determined by calling **DXGetItemSize(a)**.

DXQueryConstantArray considers both Constant Array and Regular Array Objects with zero delta values to be constant.

The value stored in **a** may also be accessed by a call to **DXGetConstantArrayData**. This call returns a pointer to the internal memory of **a**, providing a *read-only* copy of the data without the need to allocate a block of memory.

Return Value

Returns **a** if **a** contains constant data or **NULL** (without setting an error code) if **a** is an Array that does not contain constant data; otherwise, returns **NULL** and sets an error code.

See Also

DXGetConstantArrayData, **DXGetItemSize**, **DXNewConstantArray**, **DXNewRegularArray**

“Constant Arrays” on page 105.

DXQueryDistantLight

Function

Returns information about a distant Light.

Syntax

```
#include <dx/dx.h>

Light DXQueryDistantLight(Light l, Vector *direction, RGBColor *color)
```

Functional Details

Determines whether **l** is a distant Light and returns in ***direction** and ***color** the information specified when the light was created. The light source is located at an infinite distance from the scene in the direction specified by **direction**.

Return Value

Returns **1** (if it is a Distant Light) or **NULL** (if it is another type) without setting an error code; otherwise, returns **NULL** and sets an error code.

See Also

DXNewDistantLight

15.8, "Light Class" on page 156.

DXQueryGridConnections

Function

Returns information about a connections grid.

Syntax

```
#include <dx/dx.h>

Array DXQueryGridConnections(Array a, int *n, int *counts)
```

Functional Details

This routine can be used to check whether a connections Array is a regular grid.

Returns **NULL** if **a** is not an Array of regular grid connections of the sort constructed by **DXMakeGridConnections** (i.e., a Mesh Array containing only terms of type Path Array). If **n** is not **NULL**, returns the number of dimensions in the grid in ***n**. If **counts** is not **NULL**, returns the number of points along each axis in the Array pointed to by **counts**.

This routine is typically used to recognize "connections" components that are fully regular and, in most cases, much simpler to handle than regular or partially regular connections.

Return Value

Returns **a** if it is a grid connections Array or returns **NULL** but does not set an error code.

See Also

DXMakeGridConnections, DXMakeGridConnectionsV, DXMakeGridPositions, DXMakeGridPositionsV, DXQueryGridPositions

"Creating Positions and Connections Grids" on page 103.

DXQueryGridPositions

Function

Returns information about a positions grid.

Syntax

```
#include <dx/dx.h>

Array DXQueryGridPositions(Array a, int *n, int *counts,
                           float *origin, float *deltas)
```

Functional Details

Returns **NULL** if **a** is not a regular grid of the sort constructed by **DXMakeGridPositions** (i.e. a Product Array containing n terms, each term being a **TYPE_FLOAT**, n-dimensional Regular Array). If **n** is not **NULL**, it returns the number of dimensions in the grid in ***n**. If **counts** is not **NULL**, it returns the number of points along each delta vector in the Array pointed to by **counts**. If **origin** is not **NULL**, it returns the **n**-dimensional origin in the Array pointed to by **origin**. If **deltas** is not **NULL**, it returns the **n n**-dimensional delta vectors in the Array pointed to by **deltas**.

This routine is commonly used to determine whether the "positions" components are fully regular and to provide an easy mechanism for accessing information that describes the regular-positions grid. The information returned by **DXQueryGridPositions** often makes it possible to process regular positions Arrays without explicit expansion.

Array handles provide a simple mechanism for accessing individual elements of a regular grid without expansion.

Return Value

Returns **a** (if **a** is a regular grid) or **NULL** (if it is not) without setting an error code.

See Also

DXCreateArrayHandle, DXMakeGridConnections, DXMakeGridConnectionsV, DXMakeGridPositions, DXMakeGridPositionsV, DXQueryGridConnections

"Creating Positions and Connections Grids" on page 103.

DXQueryHashElement

Function

Searches a hash table for an element matching a specified key.

Syntax

```
#include <dx/dx.h>

Element DXQueryHashElement(HashTable hashTable, Key searchKey)
```

Functional Details

If a hash function was provided when the table was created, that function is used to derive a long integer pseudokey from **searchKey**; otherwise, the first long integer of **searchKey** is assumed to be the pseudokey. If a compare function was provided when the hash table was created, then more than one element may be stored with the same pseudokey. **DXQueryHashElement** will use this compare function to return the element that matches the **searchKey**.

Key is defined as:

```
typedef Pointer Key;
```

Return Value

Returns the element if found or returns **NULL**, but does not set an error code.

See Also

DXCreateHash

13.5, “Hashing” on page 139.

DXQueryOriginalSizes, DXQueryOriginalMeshExtents

Function

Returns information about the size of the original Field used as the input to **DXGrow**.

Syntax

```
#include <dx/dx.h>

Field DXQueryOriginalSizes(Field f, int *positions, int *connections)
Field DXQueryOriginalMeshExtents(Field f, int *offsets, int *sizes)
```

Functional Details

Returns information about the size of the original Field used as the input to **DXGrow**. The parameter **f** names a Field that was produced by **DXGrow**. In the case of **DXQueryOriginalSizes**, if **positions** is not **NULL**, the number of positions in the original Field is returned in ***positions**. If **connections** is not **NULL**, the number of interpolation elements in the original Field is returned in ***connections**. This is particularly useful in the case of irregular data.

In the case of data defined on a regular mesh of connections, **DXQueryOriginalMeshExtents** can be used to obtain the offsets of the original Field relative to the grown Field, and the sizes of the original Field. If **offsets** is not **NULL**, the offset in each dimension of the original Field is returned in the Array pointed to by **offsets**; if **sizes** is not **NULL**, the size in each dimension of the original Field is returned in the Array pointed to by **offsets**.

Typically, data is grown so that neighborhood information is available during the calculation of a result for some original point (or connection) in the Field. The information returned by these routines identifies the portion of the grown Field that belonged to the original Field and that will remain after a later call to **DXShrink**. Thus, the caller can use this information to process only that portion of the Field that will remain after a later call to **DXShrink**.

Return Value

Returns **f** if it is a grown field or returns **NULL** (without setting an error code) if it not; otherwise, returns **NULL** and sets an error code.

See Also

`DXGrow, DXShrink`

13.4, “Growing and Shrinking Partitioned Data” on page 137.

DXQueryParameter

Function

Determines whether an Object can be converted to a specified value type.

Syntax

```
#include <dx/dx.h>

Object DXQueryParameter(Object o, Type t, int dim, int *count)
```

Functional Details

If the conversion can be performed, the number of resulting elements is returned in **`*count`** if it is not **`NULL`**.

For successful conversion, Object **`o`** must be an Array or a String. If **`o`** is an Array, then its Category must be **`CATEGORY_REAL`**, its rank must be either 0 or 1, and it must have at least 1 item contained within.

If **`dim`** is greater than 1, then **`o`**’s rank must be 1 and its shape must match **`dim`** in order for this conversion to be successful. If **`dim`** is either 0 or 1, then both rank 0 and rank 1 shape 1 Arrays will match in size.

Once it is known that the sizes match, the Array’s Type is examined to determine whether it can be converted to the Type specified by **`t`**. In general, smaller types can be converted to larger types in the following hierarchies: **`TYPE_BYTE, TYPE_SHORT, TYPE_INT, TYPE_FLOAT, TYPE_DOUBLE, TYPE_UBYTE, TYPE_USHORT, TYPE_UINT`**.

Signed and unsigned types of the same size (e.g., **`TYPE_BYTE`** and **`TYPE_UBYTE`**) cannot be converted, nor can a signed type ever be converted to an unsigned type. Unsigned types, however, can be converted to larger signed types (e.g., **`TYPE_UBYTE`** to **`TYPE_SHORT`**).

If **`o`** is a String, **`t`** must be **`TYPE_STRING`** and **`dim`** must be either 0 or 1. If **`dim`** is 0, the String contained in **`o`** must contain only a single character.

Return Value

Returns **`o`** if the conversion can be made or returns **`NULL`** without setting an error code.

See Also

`DXExtractFloat, DXExtractInteger, DXExtractNthString, DXExtractParameter, DXExtractString, DXQueryArrayConvert`

11.8, “Extracting Module Parameters” on page 108.

DXQueryPickCount

Function

Returns the number of picks resulting from a poke.

Syntax

```
#include <dx/dx.h>

Error DXQueryPickCount(Field picks, int poke, int *pickCount)
```

Functional Details

The input to the routine is a Field (**picks**, containing pick information) and a poke number (**poke**). The pick Field is created by the Pick tool. The pick count is returned in (***pickCount**).

Return Value

Returns **OK** or returns **NULL** and sets an error code. Errors include requesting a pick count for a nonexistent poke.

See Also

DXGetPickPoint, DXQueryPokeCount, DXQueryPickPath, DXTraversePickPath

13.6, “Pick-Assistance Routines” on page 142.

DXQueryPickPath

Function

Returns information about a pick path.

Syntax

```
#include <dx/dx.h>

Error DXQueryPickPath(Field picks, int poke, int pick,
                      int *pathLength, int **path, int *elementId, int *vertexId)
```

Functional Details

For the specified Field, poke number, and pick, this routine returns:

- the length of the pick path (in **pathLength**)
- the pick path (in **path**)
- the index of the picked element (in **elementId**)
- the index of the closest vertex of the picked element (in terms of screen space) to the poke (in **vertexId**).

The input to the routine is a Field (**picks**, containing pick information), a poke number (**poke**), and the number of the pick in that poke. The pick Field is created by the Pick tool.

Return Value

Returns **OK** or returns **NULL** and sets an error code. Errors include a nonexistent poke number and a nonexistent pick number.

See Also

DXGetPickPoint, DXQueryPickCount, DXQueryPokeCount, DXTraversePickPath

13.6, “Pick-Assistance Routines” on page 142.

DXQueryPokeCount

Function

Returns the number of pokes for a specified Field.

Syntax

```
#include <dx/dx.h>

Error DXQueryPokeCount(Field picks, int  *pokeCount)
```

Functional Details

For the specified Field, this routine returns the number of pokes (in **pokeCount**).

The input to the routine is a Field (**picks**) containing pick information. The pick Field is created by the Pick tool. The poke count is returned in (***pokeCount**).

Return Value

Returns **OK** or returns **NULL** and sets an error code.

See Also

DXGetPickPoint, DXQueryPickCount, DXQueryPickPath, DXTraversePickPath

13.6, "Pick-Assistance Routines" on page 142.

DXReadyToRun

Function

Allows an asynchronous module to signal if it is ready to execute again.

Syntax

```
#include <dx/dx.h>

Error DXReadyToRun(Pointer id)
```

Functional Details

Allows an asynchronous module to signal if it is ready to execute again. **id** is the module identifier returned by **DXGetModuleId**.

A module normally does not reexecute unless one of its inputs has changed.

DXReadyToRun is normally called from the input handler routine set up with **DXRegisterInputHandler** or from a signal handler set up with the **signal**() system call.

If Data Explorer's **Execute on Change** option is active, this call will cause the network to reexcute immediately. Otherwise, the next time the user executes the network, this module will then be executed.

It is the module writer's responsibility to determine if the network was reexecuted if one of its inputs has changed because its cached output was reclaimed and needs to be regenerated or because **DXReadyToRun** was called.

Return Value

Returns **OK**, or returns **ERROR** and sets the error code to indicate an error.

See Also

DXCheckRIH, DXCompareModuleId, DXCopyModuleId, DXGetModuleId, DXRegisterInputHandler, DXSetCacheEntry

12.11, "Asynchronous Services" on page 129.

DXReAllocate

Function

Changes the size of a previously allocated block of storage.

Syntax

```
#include <dx/dx.h>

Pointer DXReAllocate(Pointer x, unsigned int n)
```

Functional Details

Changes the size of the previously allocated block of storage pointed to by **x** by **n** bytes. The pointer **x** must have been returned from either **DXAllocate**, **DXAllocateZero**, **DXAllocateLocal**, **DXAllocateLocalZero**, or a previous call to **DXReAllocate**. If **x** is **NULL**, a global memory block is allocated as if **DXAllocate** was called. The number of bytes **n** must be greater than 0. The user is responsible for freeing the allocated space by calling **DXFree**. The block is copied if necessary, invalidating any pointers to the old storage.

Return Value

Returns a pointer to the new block, or returns **NULL** and sets the error code to indicate an error such as out of memory or a corrupted storage area.

See Also

DXAllocate, DXAllocateLocal, DXAllocateLocalZero, DXAllocateZero, DXFree, DXPrintAllocate

12.3, “Memory Allocation” on page 116.

DXReference

Function

Indicates that there is a reference to a specified Object.

Syntax

```
#include <dx/dx.h>

Object DXReference(Object o)
```

Functional Details

Indicates that there is a reference to Object **o**. The Object is guaranteed to not be deleted until this reference is released, using the **DXDelete** routine.

In general, one does not need to use **DXReference**, because the Data Explorer library routines such as **DXSetMember** and **DXSetComponentValue** manage an Object’s reference count themselves.

Objects returned to the executive by modules should not be referenced. Objects returned to the executive are considered “owned” by the executive, and so it references these Objects.

Return Value

Returns **o**, or returns **NULL** and sets the error code to indicate an error.

See Also

DXDelete, DXSetCacheEntry, DXUnreference

“Object Routines” on page 119.

DXRegisterInputHandler

Function

Assigns a handler routine for input coming from an open file descriptor.

Syntax

```
#include <dx/dx.h>

Error DXRegisterInputHandler(Error (*proc)(int, Pointer), int fd, Pointer arg)
```

Functional Details

Assigns a handler routine for input coming from an open file descriptor. This function associates the routine **proc** with the file descriptor **fd** and may be used to accept input from a socket. When any input is available on **fd**, the routine **proc** is called and passed **fd** and **arg**.

The file descriptor is checked for input between module executions. Input from **fd** does not interrupt modules; **fd** is not checked for input before returning from DXRegisterInputHandler.

Calling DXRegisterInputHandler with **proc=NULL** unregisters the handler.

Return Value

Returns **OK**, or returns **ERROR** and sets the error code to indicate an error.

See Also

DXReadyToRun

12.11, "Asynchronous Services" on page 129.

DXRemove

Function

Deletes a component from a Field.

Syntax

```
#include <dx/dx.h>

Object DXRemove(Object o, char *name)
```

Functional Details

Deletes a component of the specified **name**, for each Field in Object **o**. Object **o** can be a single Field or any Object that can contain Fields (for example, Group or Series). For a single Field, this is equivalent to calling **DXDeleteComponentValue**.

Return Value

It is an error if no components of the specified **`name`** are found in any of the Fields of **`o`**. Returns **`o`** on success, or returns **`NULL`** and sets the error code to indicate an error.

See Also

`DXDeleteComponentValue, DXExists, DXExtract, DXInsert, DXRename, DXReplace, DXSwap`

11.10, "Component Manipulation" on page 110.

DXRename

Function

Renames a component in a Field.

Syntax

```
#include <dx/dx.h>

Object DXRename(Object o, char *oldname, char *newname)
```

Functional Details

Renames a component of the specified **`oldname`** to **`newname`**, for each Field in Object **`o`**. Object **`o`** can be a single Field or any Object that can contain Fields (for example, Groups or Series). If a **`newname`** already exists in a Field, it is replaced.

Return Value

It is an error if no components of the specified **`name`** are found in any of the Fields of **`o`**. Returns **`o`** on success, or returns **`NULL`** and sets the error code to indicate an error.

See Also

`DXExists, DXExtract, DXGetComponentValue, DXInsert, DXRemove, DXReplace, DXSetComponentValue, DXSwap`

11.10, "Component Manipulation" on page 110.

DXRender

Function

Renders an Object into an image.

Syntax

```
#include <dx/dx.h>

Field DXRender(Object o, Camera c, char *format)
```

Functional Details

Renders all objects in **o** using the Camera defined by **c**. It returns a new image Field containing the result. This routine performs the transformation, shading, and the tiling steps.

If **format** is specified as **NULL**, a generic floating-point image is created; this is the most flexible format with respect to processing by other modules. Alternatively, **format** may be specified as a character string identifying a hardware-specific format that may be used only for display on a particular hardware device. **format** currently must begin with "X" for an image that will be displayed on an X server image window, or "FB" for an image that will be displayed on an IBM 7246 Video Controller. Any other string will generate a generic floating-point image.

A renderable Field must have at least a "positions" and a "colors" component. For anything other than scattered points, a "connections" component is needed. Many other components and attributes affect the output of **DXRender**. For additional details on the rendering process, see Chapter 15, "Rendering" on page 149.

Return Value

Returns the image, or returns **NULL** and sets the error code to indicate an error.

See Also

DXApplyTransform, DXGetCameraMatrix, DXSetAttribute

Chapter 15, "Rendering" on page 149.

DXReplace

Function

Adds a component from one Field to another.

Syntax

```
#include <dx/dx.h>

Object DXReplace(Object o, Object add, char *src, char *dst)
```

Functional Details

Adds a component from one Field to another. For each Field in Object **o**, the **src** component of the corresponding Field in Object **add** is placed in the Field as the **dst** component.

Objects **o** and **add** can be single Fields or any Object that can contain Fields (for example, Groups or Series). If they are anything other than simple Fields, the Object hierarchies must match exactly.

Objects **o** and **add** can be the same object.

Object **add** can be an Array or an Object where each Field in **o** corresponds to an Array in **add**. In this case, the Array is added as the **dst** component of the Field and **src** is not required.

Return Value

It is an error if no components of name **`src`** are found in any of the Fields of **`add`**. Returns **`o`** on success, or returns **`NULL`** and sets the error code to indicate an error.

See Also

`DXExists`, `DXExtract`, `DXGetComponentValue`, `DXInsert`, `DXRemove`, `DXRename`, `DXSetComponentValue`, `DXSwap`

11.10, "Component Manipulation" on page 110.

DXResetError

Function

Resets the error state.

Syntax

```
#include <dx/dx.h>

void DXResetError()
```

Functional Details

Resets the error state. This should be used after correcting an error so that subsequent queries of the error state do not return an incorrect indication.

Return Value

None.

See Also

`DXGetError`, `DXSetError`

12.1, "Error Handling and Messages" on page 114

DXRGB

Function

Constructs an RGB color structure with the given components.

Syntax

```
#include <dx/dx.h>

RGBColor DXRGB(double r, double g, double b)
```

Functional Details

Fills in all three members of an RGB Color structure with the values **`r`**, **`g`**, and **`b`** at one time. RGB Colors are typically scaled between 0.0 and 1.0.

An **`RGBColor`** is defined as follows:

```
typedef struct rgbcolor {
   float r, g, b;
} RGBColor;
```

Return Value

Returns the RGB color value.

See Also

DXColorNameToRGB

“Colors” on page 125.

DXRibbon

Function

Produces a ribbon of the given width from a path or group of paths.

Syntax

```
#include <dx/dx.h>

Object DXRibbon(Object o, double width)
```

Functional Details

Produces a ribbon of the width **width** from a path or group of paths in Object **o**. The ribbon is perpendicular to the normal and parallel to the tangent at each point on the path, where the normals are provided by the “normals” component if present or approximated from the path otherwise. The normals (given or computed) are translated to the generated vertices and associated with the ribbon for shading. If the direction of the normals changes too rapidly, breaks in the ribbon may occur.

For additional details on ribbons, see Ribbon in *IBM Visualization Data Explorer User’s Reference*.

Return Value

Returns the ribbon, or returns **NULL** and sets the error code to indicate an error.

See Also

DXTube

14.3, “Path Operations” on page 146.

DXRotateX, DXRotateY, DXRotateZ, DXScale, DXTranslate, DXMat

Function

Manipulate Transform matrices.

Syntax

```
#include <dx/dx.h>

Matrix DXRotateX(Angle angle)
Matrix DXRotateY(Angle angle)
Matrix DXRotateZ(Angle angle)
Matrix DXScale(double x, double y, double z)
Matrix DXTranslate(Vector v)
Matrix DXMat(double a, double b, double c,
             double d, double e, double f,
             double g, double h, double i,
             double j, double k, double l)
```

Functional Details

Manipulate Transform matrices. **DXRotateX, DXRotateY,** and **DXRotateZ** return a Matrix that specifies a rotation about the *x*, *y*, or *z* axis by angle **angle** in radians.

DXScale returns a Matrix that specifies a scaling by amounts **x**, **y**, and **z** along the *x*, *y*, and *z* axes.

DXTranslate returns a Matrix that specifies a translation by **v**.

DXMat returns a Matrix with the specified components.

A **Matrix** is defined as follows:

```
typedef struct matrix {
    /* xA + b */
    float A[3][3];
    float b[3];
} Matrix;
```

An **Angle** is in radians and is defined as follows:

```
typedef double Angle;
```

Return Value

Returns the resulting Matrix.

See Also

DXAdjointTranspose, DXApply, DXApplyTransform, DXConcatenate, DXDeterminant, DXInvert, DXNewXform, DXTranspose

“Transformation Matrices” on page 126.

DXSaveInvalidComponent

Function

Creates a new invalid-component Array containing the information stored in an invalid-component handle, and stores it in a given field.

Syntax

```
#include <dx/dx.h>

Error DXSaveInvalidComponent(Field field, InvalidComponentHandle handle)
```

Functional Details

The new Array is stored in **field** under the component name given when **handle** was created. The invalid data may be referential or dependent, and the invalid-data component will receive an appropriate “dep” or “ref” attribute.

This routine deletes existing invalid components.

Return Value

Returns **OK** or returns **ERROR** and sets an error code.

See Also

DXCreateInvalidComponentHandle, DXFreeInvalidComponentHandle

13.3, “Invalid Data” on page 133.

DXScalarConvert

Function

Converts the contents of an Array into scalar floating-point values.

Syntax

```
#include <dx/dx.h>

Array DXScalarConvert(Array a)
```

Functional Details

Creates a new Array with the contents of Array **a** converted to scalar floating-point values, using the same conversion routines as **DXStatistics**.

Return Value

Returns a copy of the converted Array or returns **a** if it is already scalar float; otherwise, returns **NULL** and sets an error code.

See Also

DXStatistics

11.8, “Extracting Module Parameters” on page 108.

DXSetAllInvalid

Function

Sets all elements invalid.

Syntax

```
#include <dx/dx.h>

Error DXSetAllInvalid(InvalidComponentHandle handle)
```

Functional Details

Sets the validity state of all elements in invalid-component handle **handle** to **DATA_INVALID**.

Return Value

Returns **OK** or returns **ERROR** and sets an error code.

See Also

DXCreateInvalidComponentHandle, DXFreeInvalidComponentHandle

13.3, “Invalid Data” on page 133.

DXSetAllValid

Function

Sets all elements valid.

Syntax

```
#include <dx/dx.h>

Error DXSetAllValid(InvalidComponentHandle handle)
```

Functional Details

Sets the validity state of all elements in invalid-component handle **handle** to **DATA_VALID**.

Return Value

Returns **OK** or returns **ERROR** and sets an error code.

See Also

`DXCreateInvalidComponentHandle`, **`DXFreeInvalidComponentHandle`**

13.3, “Invalid Data” on page 133.

DXSetAttribute, DXDeleteAttribute

Function

Add or remove a named attribute from an Object.

Syntax

```
#include <dx/dx.h>

Object DXSetAttribute(Object o, char *name, Object value)
Object DXDeleteAttribute(Object o, char *name)
```

Functional Details

If **`value`** is not **`NULL`**, **`DXSetAttribute`** adds an attribute/value pair to Object **`o`**. The reference count of the attribute Object **`value`** is incremented. For **`DXSetAttribute`**, **`name`** specifies the name of the associated attribute and **`value`** represents its value. If **`name`** specifies an attribute that the Object **`o`** already has, then its value is replaced by **`value`**.

If **`value`** is **`NULL`**, the attribute referred to by **`name`** is removed from the Object **`o`**, if it exists.

`DXDeleteAttribute` deletes the attribute.

Return Value

Returns **`o`** or returns **`NULL`** without setting an error code.

See Also

`DXSetComponentAttribute`, **`DXSetFloatAttribute`**, **`DXSetIntegerAttribute`**, **`DXSetStringAttribute`**

“Object Routines” on page 119.

DXSetBackgroundColor, DXGetBackgroundColor

Function

Operate on a scene background color.

Syntax

```
#include <dx/dx.h>

Camera DXSetBackgroundColor(Camera c, RGBColor *background)
Camera DXGetBackgroundColor(Camera c, RGBColor background)
```

Functional Details

DXSetBackgroundColor sets a scene background color for use at render time. **DXRender** applies **background** to all pixels at the start of rendering so that unobscured pixels will have this color in the final image. The unset default background color is black, {0,0,0}.

DXGetBackgroundColor returns the value of the camera background color in ***background**. **DXRender** initializes the output image to this color when using camera **c**.

Return Value

Return **c** or return **NULL** and set an error code.

See Also

DXNewCamera, DXRender

15.7, "Camera Class" on page 155.

DXSetCacheEntry, DXSetCacheEntryV

Function

Set a cache entry.

Syntax

```
#include <dx/dx.h>

Error DXSetCacheEntry(Object out, double cost,
                      char *function, int key, int n, ...)
Error DXSetCacheEntryV(Object out, double cost,
                       char *function, int key, int n, Object *in)
```

Functional Details

Create or alter a cache entry to store a reference to **out**. The cache entry is indexed by a key created from **function**, **key**, **n**, and the Objects in the Array **in**. The parameter **function** makes the key unique to the caller; **key** allows the caller to have multiple cache entries with the same **function**, and **n** and **in** allow the cache entry to be related to the Objects that were used to create it. Setting a cache entry to **NULL** removes the entry and deletes the Object.

The **out** parameter must be a Data Explorer Object. Private Objects may be used to store arbitrary user data in the cache.

The entry may be automatically deleted at any time because of memory constraints unless **cost** is set to a value equal to or greater than **CACHE_PERMANENT**.

Notes:

1. Because Data Explorer modules follow pure function semantics, the cache should *not* be used to store a state that affects the output of the module. A module must always be able to recreate the Object from the same set of inputs; the cache should be used only as an optimization tool.
2. On a multiprocessor machine, processor local information should *not* be stored in the cache, since its contents may be retrieved on another processor.
3. The cache is local to one machine and cannot be used to communicate information between modules on different machines when running in distributed mode.

Because Objects in the cache are candidates for deletion at any time, **DXReference** should be called before caching an Object if that Object is to be used later.

If you have called DXReference on the Object before putting it in the cache, call **DXDelete** when the Object is no longer being used; the latter call will not delete the Object from the cache. (To delete an Object from the cache, set the cache entry to **NULL** with **DXSetCacheEntry**.)

Return Value

Return **OK** or return **NULL** and set an error code.

See Also

DXDelete, DXGetCacheEntry, DXGetCacheEntryV, DXGetObjectTag, DXNewPrivate, DXReference

12.5, “Cache” on page 121.

DXSetClippedObjects

Function

Sets the Object to be rendered and the Object with which to clip it during the rendering process.

Syntax

```
#include <dx/dx.h>

Clipped DXSetClippedObjects(Clipped c, Object render, Object clipping)
```

Functional Details

Given an existing Clipped Object **c**—consisting of an Object to be rendered (**render**) and an Object to do the clipping (**clipping**)—this routine replaces either or both. If **render** is not **NULL**, the Object to be clipped is replaced by **render**. If **clipping** is not **NULL**, the clipping Object is replaced by **clipping**.

Return Value

Returns **c** or returns **NULL** and sets an error code.

See Also

DXGetClippedInfo, DXNewClipped

15.6, "Clipped Class" on page 155.

DXSetComponentAttribute

Function

Adds or removes a named attribute from a component of a Field.

Syntax

```
#include <dx/dx.h>

Field DXSetComponentAttribute(Field f, char *name, char *attribute, Object value)
```

Functional Details

Adds an attribute/value pair to the component if the component **name** exists in the Field **f**, and **value** is not **NULL**. **attribute** specifies the name of the associated attribute and **value** represents its value. If **attribute** specifies an attribute that the component already has, then its value is replaced by **value**.

If **value** is **NULL**, then the attribute referred to by **attribute** is removed from the component if it exists.

Return Value

Returns **f** on success; returns **NULL** and does not set an error code if the component specified by **name** does not exist; returns **NULL** and sets an error code if **f** is not a Field.

See Also

DXGetComponentAttribute, DXSetAttribute, DXSetFloatAttribute, DXSetIntegerAttribute, DXSetStringAttribute

11.1, “Field Class” on page 97.

DXSetComponentValue

Function

Adds a component to a Field.

Syntax

```
#include <dx/dx.h>

Field DXSetComponentValue(Field f, char *name, Object value)
```

Functional Details

Sets the **name** component of Field **f** to **value**. If **name** is **NULL**, **value** can be accessed only by **DXGetEnumeratedComponentValue**. If **value** is **NULL**, the **name** component will be deleted.

When **DXSetComponentValue** overwrites an existing component, all attributes associated with the prior value are copied to the new **value** and they supersede any attributes already attached to the new **value**. If this result is not the one desired, the earlier component value should be removed prior to setting the new one.

Components of Fields are typically Arrays and contain geometrical and topological information and associated data. These components are interrelated (e.g., an association of data with either the points defined in the “positions” component or the elements defined in the “connections” component). Their relationships are specified through attributes, which should be set as the components are inserted into the Field. After all components are inserted, call **DXEndField**, which will add any additional attributes (and ancillary components) that are necessary.

Return Value

Returns **f** or returns **NULL** and sets an error code.

See Also

DXDeleteComponent, DXGetComponentValue, DXGetEnumeratedComponentValue, DXNewField, DXSetComponentAttribute

11.1, “Field Class” on page 97.

DXSetConnections

Function

Assigns a specified Array as the “connections” component of a specified Field.

Syntax

```
#include <dx/dx.h>

Field DXSetConnections(Field f, char *type, Array a)
```

Functional Details

This routine serves as a shortcut to simplify the installation of a “connections” component to a Field. It is equivalent to a call to **DXSetComponentValue** to insert the component into the Field, followed by **DXSetComponentAttribute** to set the component “element type.”

Return Value

Returns **f** or returns **NULL** and sets an error code.

See Also

DXGetConnections, DXSetComponentAttribute, DXSetComponentValue

“Connections” on page 107.

DXSetElementInvalid

Function

Sets the validity state of a specified element.

Syntax

```
#include <dx/dx.h>

Error DXSetElementInvalid(InvalidComponentHandle handle, int index)
```

Functional Details

Sets the validity state of element **index** in the invalid-component handle **handle** to **DATA_INVALID**.

Return Value

Returns **OK** or returns **ERROR** and sets an error code.

See Also

DXCreateInvalidComponentHandle, DXFreeInvalidComponentHandle

13.3, “Invalid Data” on page 133.

DXSetElementValid

Function

Sets the validity state of a specified element.

Syntax

```
#include <dx/dx.h>

Error DXSetElementValid(InvalidComponentHandle handle, int index)
```

Functional Details

Sets the validity state of element **index** in the invalid-component handle **handle** to **DATA_VALID**.

Return Value

Returns **OK** or returns **ERROR** and sets an error code.

See Also

DXCreateInvalidComponentHandle, DXFreeInvalidComponentHandle

13.3, "Invalid Data" on page 133.

DXSetEnumeratedMember

Function

Adds an Object to a Group by index.

Syntax

```
#include <dx/dx.h>

Group DXSetEnumeratedMember(Group g, int n, Object value)
```

Functional Details

Sets the value of the **n**th member of Group **g** to **value**. The parameter **n** must refer to an existing member of the Group or to the first nonexistent member. That is, the indices of Group members must always be contiguous starting at 0. If **value** is **NULL**, the **n**th member of **g** will be deleted. If **g** is typed, the Object to be added, **value**, must be the same type as **g**.

Return Value

Returns **g** or returns **NULL** and sets an error code.

See Also

DXGetEnumeratedMember, DXGetMemberCount, DXNewGroup, DXSetMember

"Generic Operations" on page 98.

DXSetError, DXErrorReturn, DXErrorGoto

Function

Set an error code and an explanatory message.

Syntax

```
#include <dx/dx.h>

Error DXSetError(ErrorCode e, char *message, ...)
#define DXErrorReturn(e,s) {DXSetError(e,s); return ERROR;}
#define DXErrorGoto(e,s) {DXSetError(e,s); goto error;}
```

Functional Details

Modules that return **ERROR** should in many cases also set the error code and error message. **DXSetError** provides the interface to do this, and needs to be called only once for each error.

If a Data Explorer function returns an error and sets an error code, the caller should usually return **ERROR** without using one of these routines, since the error message set by the Data Explorer function will usually be more informative. If the error code is set by the Data Explorer function and the calling program proceeds without returning **ERROR**, **DXResetError** should be called.

If the Data Explorer function does not set the error code, it is the calling function's responsibility to do so using one of these routines.

The **message** may be a **printf** format string, in which case additional arguments as required by the format string must be specified. Messages beginning with the pound sign, **#**, are reserved for system use.

Additional information can be added to an error message using the **DXAddMessage** function.

DXErrorReturn calls **DXSetError** and returns from the caller with return value **ERROR**.

DXErrorGoto calls **DXSetError** and goes to the label “error.” This is useful when some clean-up activity is required (e.g., freeing allocated memory). It is the caller's responsibility to provide the “error” label in the code.

The error code **e** must be one of the following:

```
ERROR_ASSERTION        ERROR_INTERNAL        ERROR_NO_CAMERA
ERROR_BAD_CLASS        ERROR_INVALID_DATA    ERROR_NO_MEMORY
ERROR_BAD_PARAMETER    ERROR_MISSING_DATA    ERROR_NOT_IMPLEMENTED
ERROR_BAD_TYPE                               ERROR_UNEXPECTED
```

Return Value

Always return **ERROR**.

See Also

`DXAddMessage`, `DXMessage`, `DXResetError`, `DXWarning`

12.1, “Error Handling and Messages” on page 114.

DXSetErrorExit

Function

Determines the action taken when DXSetError is called in a stand-alone program.

Syntax

```
#include <dx/dx.h>

Void DXSetErrorExit(int level);
```

Functional Details

Valid arguments for **`level`** are:

`0` = store error message. Use **`DXPrintError()`** to print when ready.

`1` = print error message and return.

`2` = print error message and exit.

This routine is intended for those using Data Explorer library routines in their own programs, and for use only in stand-alone programs. By default, **`DXInitModules`** sets **`level`** to 1 (one). **`DXSetErrorExit`** affects the behavior of **`DXSetError`** only outside those built-in Data Explorer modules called with **`DXCallModule`**.

Return Value

No return value.

See Also

`DXGetErrorExit`

DXSetFloatAttribute

Function

Adds a named attribute with a floating-point value to an Object.

Syntax

```
#include <dx/dx.h>

Object DXSetFloatAttribute(Object o, char *name, double x)
```

Functional Details

Creates an Array containing the floating-point value **x** and then adds an attribute/value pair to the Object **o**. **name** specifies the name of the attribute and **x** represents its value. If **name** specifies an attribute that the Object **o** already has, then its previous value is replaced.

Return Value

Returns **o** or returns **NULL** and sets an error code.

See Also

DXGetFloatAttribute, **DXSetAttribute**, **DXSetComponentAttribute**, **DXSetIntegerAttribute**, **DXSetStringAttribute**

"Object Routines" on page 119.

DXSetGroupType, DXSetGroupTypeV

Function

Associate a type with a Group.

Syntax

```
#include <dx/dx.h>

Group DXSetGroupType(Group g, Type t, Category c, int rank, ...)
Group DXSetGroupTypeV(Group g, Type t, Category c, int rank, int *shape)
```

Functional Details

Associate a type **t**, category **c**, **rank**, and **shape**, (hereafter referred to as simply type), with Group **g**. When the Group type is set, all current members are checked for type, and all members added subsequently are checked for type. The type of a Group may be retrieved by **DXGetType**.

The type is one of the following:

TYPE_BYTE	**TYPE_HYPER**	**TYPE_SHORT**
TYPE_UBYTE	**TYPE_INT**	**TYPE_USHORT**
TYPE_DOUBLE	**TYPE_UINT**	**TYPE_STRING**
TYPE_FLOAT		

The category is either **CATEGORY_REAL** or **CATEGORY_COMPLEX**.

Array Objects are always typed. Fields are typed if they contain a “data” component; their type is the same as that of the “data” component. Series, MultiGrids, and Composite Fields are typed if they contain typed Fields. **DXSetGroupType** may be used to explicitly type generic Groups. If typed, all Fields contained in the Group must match the type. Other Objects do not contain type information. **DXSetGroupType** needs to be used if the member Fields are manipulated so that the type of their “data” components, and therefore the Field type, changes.

Return Value

Returns **g** or returns **NULL** and sets an error code.

See Also

DXNewGroup, DXSetEnumeratedMember, DXSetMember

“Generic Operations” on page 98.

DXSetIntegerAttribute

Function

Adds a named attribute with an integer value to an Object.

Syntax

```
#include <dx/dx.h>

Object DXSetIntegerAttribute(Object o, char *name, int x)
```

Functional Details

Creates an Array containing the integer value **x** and then adds an attribute/value pair to the Object **o**. **name** specifies the name of the attribute and **x** represents its value. If **name** specifies an attribute that the Object **o** already has, then its previous value is replaced.

Return Value

Returns **o** or returns **NULL** and sets an error code.

See Also

DXGetIntegerAttribute, DXSetAttribute, DXSetComponentAttribute, DXSetFloatAttribute, DXSetStringAttribute

“Object Routines” on page 119.

DXSetMember

Function

Adds a member to a Group.

Syntax

```
#include <dx/dx.h>

Group DXSetMember(Group g, char *name, Object value)
```

Functional Details

Adds **value** as a member of Group **g**. The **name** may be **NULL**, in which case a new member is added that may be accessed only using **DXGetEnumeratedMember**. If **name** is the same as the name of an existing member, then the new member will have the same index in the Field as the old member and the value of that member is overwritten with the new value. Setting a member to **NULL** deletes the member.

If the Group is typed, and the new Object **value** is typed, then **value** must be of the same type as the Group. This is generally the case for Series, Composite Fields, and MultiGrids. The first time **DXSetMember** is called for one of these generally typed Groups with a typed **value**, **DXSetMember** sets the type of the Group to be the type of the Object.

Return Value

Returns **g** or returns **NULL** and sets an error code.

See Also

DXGetEnumeratedMember, DXGetGroupClass, DXGetMember, DXNewGroup, DXSetEnumeratedMember, DXSetGroupType

“Generic Operations” on page 98.

DXSetMeshOffsets

Function

Sets the offset of a partition within the original Field after partitioning.

Syntax

```
#include <dx/dx.h>

MeshArray DXSetMeshOffsets(MeshArray a, int *offsets)
```

Functional Details

In the case where a Mesh Array is used to define a regular grid of connections that is a part of a partitioned Field, it is useful to know the offset of the partition within the original Field. This routine sets the offset values to **offsets** along each dimension of the mesh **a**. For **DXSetMeshOffsets**, the parameter is a pointer to an Array of integers, one for each dimension of the mesh, specifying the offset along that dimension of this partition within the original Field. **DXSetMeshOffsets** should be called by the partitioning process.

Return Value

Returns **a** or returns **NULL** and sets an error code.

See Also

DXGetMeshArrayInfo, DXGetPathOffset, DXNewMeshArray, DXNewMeshArrayV

"Mesh Arrays" on page 105.

DXSetOrthographic, DXGetOrthographic

Function

Sets and retrieves an orthographic camera view.

Syntax

```
#include <dx/dx.h>

Camera DXSetOrthographic(Camera c, double width, double aspect)
Camera DXGetOrthographic(Camera c, double width, double aspect)
```

Functional Details

A camera defines the position and orientation of the viewer, the volume of interest of the object being viewed, and the size of the image to contain the resulting view.

DXSetOrthographic defines the volume of interest for an orthographic camera **c**. This can be thought of as a box that is centered on the **to** point, with its **z** axis parallel to the **to-from** vector, and infinite in length. Its **y** axis is perpendicular to the **to-from** vector in the direction of the **up** vector. Its **x** axis is perpendicular to its **y** and **z** axes. The x and y dimensions of the box are given by the **width** and **aspect** parameters of the camera respectively, where **aspect** is defined as the ration of the height to width.

In orthographic projection, objects do not appear smaller as they get more distant, and in fact, distance between the object and viewer have no effect on the appearance of the object. The distance between the **to** and **from** points is irrelevant; only the direction is important.

DXGetOrthographic returns the **width** and **aspect** parameters of an orthographic camera **c**. If width is not **NULL**, the camera width is returned in ***width**. If aspect is not **NULL**, the camera aspect is returned in ***aspect**.

Return Value

DXSetOrthographic returns the camera or returns **NULL** and sets an error code.

DXGetOrthographic returns the parameters or returns **NULL** (if the camera is not orthographic) and sets an error code (if **c** is not a valid camera).

See Also

DXGetCameraMatrix, DXNewCamera, DXRender, DXSetPerspective, DXSetResolution, DXSetView

15.7, "Camera Class" on page 155.

DXSetPart

Function

Adds a Field to an Object.

Syntax

```
#include <dx/dx.h>

Object DXSetPart(Object o, int n, Field field)
```

Functional Details

Performs a depth-first traversal of the Object **c**, and replaces the **n**th occurrence of a subObject with class **CLASS_FIELD** with the Field **field** given. If the root of the Object given is not one of **CLASS_GROUP**, **CLASS_XFORM**, **CLASS_CLIP**, or **CLASS_SCREEN**, this function has no effect.

For applying a function to every Field in a Group, **DXProcessParts** is a more efficient interface.

The parts of a Group may be indexed by calling **DXSetPart** with successive values of **n**, starting at 0 until **NULL** is returned, provided the replacement part contains the same number of subObjects of **CLASS_FIELD**. This is because the replacement part will be traversed (and counted) in subsequent calls to **DXSetPart**.

Return Value

Returns **o** or returns **NULL** and sets an error code.

See Also

DXGetPart, DXGetPartClass, DXProcessParts

"Parts" on page 100.

DXSetPathOffset

Function

Sets the offset of a Path Array within the original Field after partitioning.

Syntax

```
#include <dx/dx.h>

PathArray DXSetPathOffset(PathArray a, int offset)
```

Functional Details

Sets the offset value for a portion of the Path Array relative to the original grid to **offset**. In the case where a Path Array **a** is used to define a regular grid of connections that is a part of a partitioned Field, it is useful to know the offset of the partition within the original Field.

Path Arrays are typically used as constituents in Mesh Arrays that define regular or partially regular connections grids of one or more dimensions. In that case, the mesh offsets of the partition within the original mesh are accessed at the Mesh Array level through calls to **DXSetMeshOffsets** and **DXGetMeshOffsets**.

Return Value

Returns **a** or returns **NULL** and sets an error code.

Returns **o** or returns **NULL** and sets an error code.

Returns **OK** or returns **ERROR** and sets an error code.

See Also

DXGetMeshOffsets, DXGetPathArrayInfo, DXNewPathArray, DXSetMeshOffsets

“Path Arrays” on page 104

DXSetPendingCmd

Function

Enters a task into a list of tasks to be run at the end of each graph (visual program) execution.

Syntax

```
#include <dx/dx.h>

Error DXSetPendingCmd(char *major, char *minor, int(*task)(Private),
                      Private data);
```

Functional Details

The task to be run (***task**) is identified by two character strings:

***major** typically contains the calling module’s ID.

***minor** typically indicates which of several discrete tasks the module has entered.

When execution of the graph is completed, the tasks in the list are executed in the order in which they are received. (If tasks are entered by two different modules, the order in which the modules run determines the order in which the tasks are called.)

When ***task** is called, it is passed ***data** as a parameter. The specified task is called at the end of each execution until it is removed from the pending-task list. Removal can be achieved by calling the routine again, with the same strings for ***major** and ***minor**, but setting ***task** to NULL.

Return Value

Returns OK or returns ERROR and sets and error code.

See Also

DXGetModuleId

12.6, "Pending Commands" on page 122.

DXSetPerspective, DXGetPerspective

Function

Set or retrieve a perspective view.

Syntax

```
#include <dx/dx.h>

Camera DXSetPerspective(Camera c, double fov, double aspect)
Camera DXGetPerspective(Camera c, float *fov, float *aspect)
```

Functional Details

A camera defines the position and orientation of the viewer, the volume of interest of the Object being viewed, and the size of the image to contain the resulting view.

DXSetPerspective defines the volume of interest of a perspective camera **c**. This is a pyramid with an apex at the **from** position, and a base centered on **to** point, perpendicular to the **to-from** vector. The width of the base is defined by the angle formed by the sides of the pyramid at the apex and the distance between the **to** and **from** points. This angle is also known as the "field of view" and is specified by the **fov** parameter.

The **fov** is defined as twice the tangent of half the angle (e.g, for a 90-degree sweep, the **fov** setting should be 2 * tan(45-degrees), or 2.0). The **fov** can also be thought of as the ratio of the width of the base to the distance from the viewer (e.g., for a view of a 20-meter wide area from a distance of 10 meters, the **fov** setting should be 20/10, or 2.0). The height is defined by width of the base times the **aspect**. With a perspective camera, objects appear smaller with increasing distance.

DXGetPerspective returns the **fov** and **aspect** parameters of a perspective camera **c**. If **fov** is not **NULL**, returns the camera fov in ***fov**. If **aspect** is not **NULL**, returns the camera aspect ratio in ***aspect**.

Return Value

DXSetPerspective returns the camera or returns **NULL** and sets an error code.

DXGetPerspective returns as follows:

- **If the object is a perspective camera:** returns the camera and the camera parameters.
- **If the object is not a perspective camera:** returns **NULL** without setting an error code.
- **If the object is not a valid camera:** returns **NULL** and sets an error code.

See Also

DXGetCameraMatrix, **DXNewCamera**, **DXRender**, **DXSetOrthographic**, **DXSetResolution**, **DXSetView**

15.7, "Camera Class" on page 155.

DXSetResolution, DXGetCameraResolution

Function

Set or retrieve the resolution of a Camera.

Syntax

```
#include <dx/dx.h>

Camera DXSetResolution(Camera c, int hres, double pix_aspect)
Camera DXGetCameraResolution(Camera c, int Xresolution, int Yresolution)
```

Functional Details

A Camera defines the position orientation of the viewer, the volume of interest of the object being viewed, and the size of the image to contain the resulting view.

The horizontal size of the image containing the resulting view is defined by the **resolution** in pixels; the vertical size is determined by the **resolution** * **aspect**. This is camera **aspect**, not **pix_aspect**; see **DXSetPerspective** and **DXSetOrthographic**. The same aspect ratio is used for both the size of the image and the volume of the interest to prevent the objects from being stretched in one of the dimensions.

If the display pixels are not square, **pix_aspect** can be used to compensate. Pixels are not square when the ratio of the number of pixels in the x and y dimensions does not equal the width:height ratio of the screen.

pix_aspect prevents objects from appearing stretched in one dimension (e.g. preventing a circle from becoming an ellipse on a screen with rectangular pixels). Pixels are assumed to be **pix_aspect** times as tall as they are wide. Most screens have square pixels and it is rarely necessary to set **pix_aspect** to a value other than 1.

DXSetResolution, for camera **c**, sets the horizontal resolution to **hres** and the pixel aspect ratio to **pix_aspect**.

DXGetCameraResolution, for camera **c**, if **Xresolution** is not **NULL**, returns the **resolution** in ***Xresolution**. If **Yresolution** is not **NULL**, returns the **resolution** * **aspect** in ***Yresolution**. Again, this is camera aspect.

Return Value

DXSetResolution returns the Camera or returns **NULL** and an error code.

DXGetCameraResolution returns the parameters or returns **NULL** and sets an error code.

See Also

`DXGetCameraMatrix`, `DXNewCamera`, `DXRender`, `DXSetOrthographic`, `DXSetPerspective`, `DXSetView`

15.7, “Camera Class” on page 155.

DXSetScreenObject

Function

Sets the Object that a screen transform is to be applied to.

Syntax

```
#include <dx/dx.h>

Screen DXSetScreenObject(Screen s, Object o)
```

Functional Details

Replaces the Object the screen transformation is to be applied to with Object **`o`**, given an existing screen object **`s`**.

Return Value

Returns **`s`** or returns **`NULL`** and sets an error code if **`s`** is not a screen object.

See Also

`DXGetScreenInfo`, `DXNewScreen`

15.5, “Screen Class” on page 154.

DXSetSeriesMember

Function

Adds an indexed member to a Series Object.

Syntax

```
#include <dx/dx.h>

Series DXSetSeriesMember(Series s, int n, double position, Object o)
```

Functional Details

Adds the Object **`o`** as the **`n`**th member (as specified by the zero-based index **`n`**) to the Series **`s`**. It also sets the Series position to the floating-point value **`position`**. A Series is intended to represent a single Field sampled across some parameter, such as time or temperature (**`position`** contains the value of this sampled parameter).

Generally, members of a Series **`s`** must be added sequentially, with indices starting at 0. If, however, an already existing index is given in **`n`**, then that member in **`s`** will be replaced by the Object **`o`**. The type of a Series is set the first time a typed member is added; subsequent members are required to match this type.

The index **n** provides a convenient mechanism for indexing the members of a Series. The position value is actually a piece of metadata to be associated with each member of the Series, which when used with the values associated with the other members, provides a means of generating intermember data (e.g., when using the Streakline module).

DXSetSeriesMember does not enforce any restrictions on the Series **position** value; duplicate or non-monotonic values can be set. However, many Realization modules require the position values to form a monotonic sequence, and if the value does represent time, some modules also require the values to be positive.

For an example of one use of Series, see “Streakline” on page 323 in *IBM Visualization Data Explorer User’s Reference*.

Return Value

Returns **NULL** and sets an error code.

See Also

DXGetMemberCount, DXGetSeriesMember, DXNewSeries, DXSetEnumeratedMember, DXSetMember, Streakline

“Series Groups” on page 99.

DXSetStringAttribute

Function

Adds a named attribute with a string value to an Object.

Syntax

```
#include <dx/dx.h>

Object DXSetStringAttribute(Object o, char *name, char *x)
```

Functional Details

Creates a String containing the string value **x** and then adds an attribute/value pair to the Object **o**. **name** specifies the name of the attribute and **x** represents its value. If **name** specifies an attribute that the Object **o** already has, then its previous value is replaced.

Return Value

Returns **o** or returns **NULL** and sets an error code.

See Also

DXGetStringAttribute DXSetAttribute, DXSetComponentAttribute, DXSetFloatAttribute, DXSetIntegerAttribute

“Object Routines” on page 119.

DXSetView, DXGetView

Function

Specify Camera position and orientation.

Syntax

```
#include <dx/dx.h>

Camera DXSetView(Camera c, Point from, Point to, Vector up)
Camera DXGetView(Camera c, Point *from, Point *to, Vector *up)
```

Functional Details

A camera defines the position and orientation of the viewer, the volume of interest of the object being viewed, and the size of the image to contain the resulting view.

The position and orientation of the view are defined by where the viewer is standing, the **from** position, where the viewer is looking, the **to** position, and the tilt of the viewer's head, the **up** vector.

The image is always in a plane perpendicular to the **from-to** vector. The **up** vector is projected onto this plane and the Object and camera (the **up** vector) are rotated around the **from-to** vector until **up** is aligned with the image **y** axis. It is not necessary that the **from-to** vector and the **up** vector be perpendicular; if they are coincident, **up** becomes undefined and the **top** of the object, relative to the image, becomes undefined.

DXSetView sets the parameters **from**, **to**, and **up** for camera **c**.

DXGetView, for camera **c**, if **from** is not **NULL**, returns in **from** the from position. If **to** is not **NULL**, it returns in **to** the to position. If **up** is not **NULL**, it returns in **up** the up vector.

Return Value

DXSetView returns **NULL** and sets an error code.

DXGetView returns the camera parameters or returns **NULL** and sets an error code.

See Also

DXGetCameraMatrix, DXNewCamera, DXRender, DXSetOrthographic, DXSetPerspective, DXSetResolution

15.7, "Camera Class" on page 155.

DXSetXformObject

Function

Sets the Object to which a transform is applied.

Syntax

```
#include <dx/dx.h>

Xform DXSetXformObject(Xform x, Object o)
```

Functional Details

Replaces the object to be transformed, given an existing Xform object **x**. If **o** is not **NULL**, then the object to be transformed contained in the Xform object **x** is replaced by **o**.

Return Value

Returns **x** or returns **NULL** and sets an error code.

See Also

DXGetXformInfo, DXNewXform

15.4, "Xform Class" on page 154.

DXShrink

Function

Removes information added to an Object by **DXGrow**.

Syntax

```
#include <dx/dx.h>

Object DXShrink(Object object)
```

Functional Details

Removes information from Object **object** added onto each partition of Composite Field members of **object** by **DXGrow**.

Each "original *component*" in **object** is renamed "*component,*" replacing any existing components of that name. To prevent this, the "original *component*" can be deleted with **DXRemove** or **DXDeleteComponent** before **DXShrink** is called.

Return Value

Returns the Object with added data removed or returns **NULL** and sets an error code.

See Also

DXDeleteComponent, DXGrow, DXRemove

13.4, "Growing and Shrinking Partitioned Data" on page 137.

DXStatistics

Function

Returns statistical information about a specified Object.

Syntax

```
#include <dx/dx.h>

Error DXStatistics(Object o, char *component, float *min, float *max,
                   float *avg,  float *sigma)
```

Functional Details

Returns statistical information about all specified components (**component**) of Object **o**. If **min** is not **NULL**, this routine returns the minimum value in ***min**. If **max** is not **NULL**, the routine returns the maximum value in ***max**. If **avg** is not **NULL**, it routine returns the average value in ***avg**. If **sigma** is not **NULL**, it returns the standard deviation in ***sigma**. The "*component* statistics" component is added to Fields for which it does not already exist. If Object **o** is an Array, the routine ignores **component** and returns statistics for the Array.

Return Value

Returns **OK** or returns **ERROR** and sets an error code.

See Also

"Standard Components" on page 107.

DXSwap

Function

Interchanges two components in a Field.

Syntax

```
#include <dx/dx.h>

Object DXSwap(Object o, char *name1, char *name2)
```

Functional Details

Interchanges the component values for each Field in Object **o**, if both components (**name1** and **name2**) exist. Object **o** can be a single Field or any Object that can contain Fields, such as Groups or Series.

Return Value

Returns **o** or returns **NULL** and sets an error code. It is an error if any Field of Object **o** contains one of the named components but not the other.

See Also

`DXExists`, `DXExtract`, `DXGetComponentValue`, `DXInsert`, `DXRemove`, `DXRename`, `DXReplace`, `DXSetComponentValue`

11.10, “Component Manipulation” on page 110.

DXTraceTime

Function

Enables or disables the accumulation of time marks.

Syntax

```
#include <dx/dx.h>

void DXTraceTime(int t)
```

Functional Details

Enables (**t** is 1) or disables (**t** is 0) the accumulation of time marks by **DXMarkTime** and **DXMarkTimeLocal** and the printing of timing messages by **DXPrintTimes**.

Return Value

None.

See Also

DXGetTime, DXMarkTime, DXMarkTimeLocal, DXPrintTimes

12.2, “Timing” on page 116.

DXTraversePickPath

Function

Returns the subObject of the current Object selected by a pick path.

Syntax

```
#include <dx/dx.h>

Object DXTraversePickPath(Object current, int index, Matrix *matrix)
```

Functional Details

Traverses a data Object to reach the picked Field, element, and vertex. Given an Object **current** (initially the root of the data Object), a pointer to a matrix **matrix** (initially identity), and a path index **index**, returns the subObject of the current Object.

When Xform Objects are encountered, the matrix associated with the Xform is concatenated onto the matrix pointed to by the **matrix** parameter (if one was passed in). When the end of the path is found (either by recognizing that the returned Object is a Field or that the returned Object is the same as the current Object), the caller is left with the picked Field and a transform carrying the coordinate system of that Field to the eye coordinate system.

Return Value

Returns the subObject or returns **`NULL`** and sets an error code.

See Also

`DXGetPickPoint, DXQueryPickCount, DXQueryPickPath, DXQueryPokeCount`

13.6, "Pick-Assistance Routines" on page 142.

DXTrim

Function

Frees space allocated in an Array beyond that required for the number of items in the Array.

Syntax

```
#include <dx/dx.h>

Array DXTrim(Array a)
```

Functional Details

Under some circumstances, more space than is necessary to hold the items added to **`a`** may have been allocated. This can happen if you have called **`DXAllocateArray`**. It can also happen when you call **`DXAddArrayData`**. This extra space can be freed by calling **`DXTrim`**. The **`DXEndField`** routine automatically calls **`DXTrim`** on all components of a Field.

Return Value

Returns **`a`** or returns **`NULL`** and sets an error code.

See Also

`DXAddArrayData, DXAllocateArray, DXEndField`

"Irregular Arrays" on page 101.

DXTube

Function

Produces a tube of specified diameter from a path or group of paths in a specified Object.

Syntax

```
#include <dx/dx.h>

Object DXTube(Object o, double diameter, int n)
```

Functional Details

The cross section of the tube is an **n**-gon in a plane parallel to the normal and perpendicular to the tangent at each point on the path, where the normals are provided by a “normals” component if present or approximated from the path otherwise. Normals to the tube are computed and associated with the tube for shading.

For more information, see “Tube” on page 356 in *IBM Visualization Data Explorer User's Reference*. See also 2.4, “Memory Management” on page 13.

Return Value

Returns the tube or returns **NULL** and sets an error code.

See Also

DXRibbon

14.3, “Path Operations” on page 146.

DXTypeCheck, DXTypeCheckV

Function

Check that an Array meets a set of specifications.

Syntax

```
#include <dx/dx.h>

Array DXTypeCheck(Array a, Type type, Category category, int rank, ...)
Array DXTypeCheckV(Array a, Type type, Category category, int rank, int *shape)
```

Functional Details

The routine returns **a** if that Array meets the specifications given by **type**, **category**, **rank**, and **shape**. Otherwise, it returns **NULL**. The shape is specified by **shape** for **DXTypeCheckV** or by the last **rank** arguments for **DXTypeCheck**. For **DXTypeCheckV**, if **shape** is **NULL**, the type, category, and rank are checked, but the shape is not.

rank specifies the rank of the items in the Array: Scalars have rank 0, vectors have rank 1, and so on.

shape has rank entries representing the list of dimensions of the structure. For rank 0 items (scalars) there is no shape. For rank 1 items (vectors) the shape is a single number corresponding to the number of dimensions. For rank 2 structures, the shape is two (2) numbers, and so on. **shape** specifies the rank of the items in the Array: the number of dimensions in each item of the Array. Shape has entries where each entry represents the size of the item in that dimension.

The type is one of the following:

TYPE_BYTE	**TYPE_HYPER**	**TYPE_SHORT**
TYPE_UBYTE	**TYPE_INT**	**TYPE_USHORT**
TYPE_DOUBLE	**TYPE_UINT**	**TYPE_STRING**
TYPE_FLOAT		

The category is either **CATEGORY_REAL** or **CATEGORY_COMPLEX**.

Return Value

Return **a** or returns **NULL** (e.g., if the type does not match).

See Also

DXGetArrayInfo

11.3, “Array Class” on page 101.

DXTypeSize, DXCategorySize

Function

Return size information.

Syntax

```
#include <dx/dx.h>

int DXTypeSize(Type t)
int DXCategorySize(Category c)
```

Functional Details

DXTypeSize returns the size in bytes of a variable of type **t**.
DXCategorySize returns the size multiplier for category **c**.

For a variable of type **t** and category **c**, the size in bytes is **DXTypeSize(t) * DXCategorySize(c)**.

Note: This is the size of a single item of that type (e.g., a single component of a 3-vector). **DXGetItemSize**, in contrast, returns the size in bytes of the 3-vector.

The type is one of the following:

The category is either **CATEGORY_REAL** or **CATEGORY_COMPLEX**.

Return Value

DXTypeSize returns the size in bytes. **DXCategorySize** returns the multiplier in bytes.

See Also

DXGetItemSize

“Setting Data Types” on page 120.

DXUnreference

Function

Removes a reference from an Object without deleting it.

Syntax

```
#include <dx/dx.h>

Error DXUnreference(Object o)
```

Functional Details

This routine is not normally used by module writers. **DXDelete** should be called to remove a reference to an Object **o**.

DXUnreference allows an Object's reference count to be decremented to 0 without releasing its memory, which is not desirable under normal circumstances. (See 2.4, "Memory Management" on page 13.)

Return Value

Returns **OK** or returns **ERROR** and sets an error code.

See Also

DXDelete, DXReference

"Object Routines" on page 119.

DXUnsetGroupType

Function

Unsets the type associated with a specified Group.

Syntax

```
#include <dx/dx.h>

Group DXUnsetGroupType(Group g)
```

Functional Details

When the Group type is set, all current members are checked for type, and all members added subsequently are checked for type. **DXUnsetGroupType** will turn off this type checking, and the Group will have no type.

Array Objects are always typed. Fields are typed if they contain a "data" component; their type is the same as that of the "data" component. Series, MultiGrids, and Composite Fields are typed if they contain typed members. Generic Groups may be typed by explicitly calling **DXSetGroupType**. If typed, all members contained in the Group must match the type. Other Objects do not contain type information.

Return Value

Returns **`g`** or returns **`NULL`** and sets an error code.

See Also

`DXNewGroup, DXSetGroupType`

“Generic Operations” on page 98.

DXValidPositionsBoundaryBox

Function

Computes the boundary box of the valid positions of an Object `o`.

Syntax

```
#include <dx/dx.h>

DXValidPositionsBoundaryBox(Object o, Point *box)
```

Functional Details

If `o` contains any invalid positions, this routine adds (to `o` or to any of its descendents that are Fields) a “valid box” component consisting of an Array of 2^d points that are the corners of a boundary box (where *d* is the dimensionality of the data).

`Point` is defined as follows:

```
typedef struct point {
    float x,y,z;
} Point, Vector;
```

Notes:

1. For data of dimensionality three or less, the routine returns (in the Array pointed to by `box`) the eight corner points. For dimensionalities less than three, the extra dimensions of the box returned by the routine are treated as zero.
2. The boundary box returned by the routine is determined by combining the boundary boxes of all Fields contained in `o`.
3. Transformations are considered in computing the boundary box, but clipping objects are not. And the boundary box is not guaranteed to be the tightest possible.

Return Value

Returns `o` or returns `NULL` and might or might not set an error code, depending on the input. For example, it does not set an error code if a boundary box cannot be defined for the given input.

See Also

`DXBoundingBox`, `DXChangedComponentValues`, `DXChangedComponentStructure`, `DXEmptyField`, `DXEndField`, `DXEndObject`, `DXNeighbors`, `DXStatistics`

12.1, “Error Handling and Messages” on page 114.

DXWarning

Function

Presents a warning message to the user.

Syntax

```
#include <dx/dx.h>

void DXWarning(char *message, ...)
```

Functional Details

The **message** may be a **printf** form string, in which case additional arguments may be needed. The message string should not contain new-line characters, because the **DXWarning** routine formats the message in a manner appropriate to the output medium. For terminal output, this includes prefixing the message with the processor identifier and appending a new line character.

Return Value

None.

See Also

DXSetError, DXErrorReturn, DXErrorGoto

12.1, "Error Handling and Messages" on page 114.

Glossary

This glossary defines important abbreviations and terms. It includes information from *IBM Dictionary of Computing* (ZC20-1699).

A

Array. An ordered list of data items of the same type represented by an Array Object. Arrays are either irregular or compact. See also *compact Array, irregular Array*.

attribute. A characteristic of an Object. Objects can have attributes that are indexed by a string name and have a value that is an Object. See also *component attribute*.

C

cache. The cache service provides various parts of the system with a means to store the results of computations for later reuse. Each cache entry is uniquely identified by a string function name, an integer key (used by the executive to store multiple outputs for a single module), the number of input parameters, and the set of input parameter values.

camera. A camera defines the position and orientation of the viewer, the volume of interest of the object being viewed, and the size of the image to contain the resulting view.

canvas. The large open area of the VPE window that is used to build and edit visual programs.

clipping plane. A plane that divides a 3-dimensional volume into a region that is rendered and a region that is not rendered. This allows the inside of an object to be seen.

compact array. Any one of five varieties of compact encoding of array data. The varieties are:

- regular
- path
- product
- mesh
- compact.

component. A basic part or classification of a Field; each component is indexed by a string and its value is an Object. See also *component attribute*.

component attribute. A characteristic of a component. Components of a Field can have attributes that are indexed by a string name and have a value that is an Object.

Composite Field. A group of Fields that together is treated a single Field. This type of Field is useful, for example, for certain kinds of simulation data that are represented by disjoint grids. A Composite Field is a collection of compatible Fields, all having a “positions” component of the same dimensionality, “data” components of the same type, and “connections” with the same “element type.” It is required that members be disjoint and abutting.

connections. A component of a Field, specifying how data points in that Field are joined together.

constant Array. An Array with a constant value.

contour. Lines on a surface that connect points with the same data value.

D

dependent. A description of a component attribute. One component is said to be dependent on another if the items in the component arrays are in one-to-one correspondence with each other.

E

executive. The component of the Data Explorer system that manages the execution of modules specified using the scripting language. This term is often used to refer to the entire server portion of the Data Explorer client-server model, including the executive, modules, and data management components.

export. To write an Object to a specified Data Explorer-format external data file. For example, importing an ASCII file can be slow. You can export the data in a binary format for faster access. You can also import data, operate on it, and then export the transformed data for future use.

F

Field. A self-contained collection of information necessary to represent scientific data. A Data Explorer Field typically is made up of a series of components and other information as required. It includes the data itself in the form of a “data” component, a set of sample points in the form of a “positions” component, optionally, a set of interpolation elements in the form of a “connections” component, and other information as needed.

filter. A program that converts data to a Data Explorer format.

flat shading. A shading model in which each polygon of an object is shaded using a single intensity value. Contrast with *Gouraud shading*.

fork-join parallelism. A programming concept that allows parallel processing. The fork statement splits a single computation into multiple independent computations. The join statement recombines two or more concurrent computations into one.

G

glyph. A figure that is mapped to a particular variable. The length, angle, or other attribute of the figure relates to the value of the particular variable.

Gouraud shading. Also called intensity interpolation shading. A shading model in which the intensity of values of incident illumination on a polygon are interpolated from intensity values at the vertices of the polygon. Contrast with *flat shading*.

Group. A collection of Objects.

grow. Add information from neighboring partitions to a Composite Field. Generally necessary only in a parallel environment.

H

handle. A data structure that is a temporary local identifier for an Object.

hash table. A table of Objects that is accessed with a search key (the hash value).

hidden. A module input parameter that does not appear by default in the module configuration dialog box. Hidden parameters are typically those less commonly used in visual programs.

I

Image window. IBM Data Explorer window that displays the image generated by a visual program.

import. To read in an external data file.

interactor. A Data Explorer device used to manipulate data in order to change the visual image produced by a program.

See also *interactor stand-in*.

interactor stand-in. An icon used in the visual program editor window to represent an interactor. Stand-ins are named after the type of data they accept. The five types are:

- integer
- scalar
- string
- selector
- value.

interpolation element. An interpolation element provides a means to interpolate between a specified set of sample points. The set of interpolation elements for a particular Field constitutes the “connections” component.

invalid positions and connections. Marked positions and connections that are ignored by modules.

irregular Array. An Array Object in which the data is stored explicitly, as opposed to a compact Array.

isosurface. A surface in 3-dimensional space that connects points with the same data value.

item. A single piece of data in an Array Object.

J

join. An operation that merges two or more computation paths.

L

light. An Object that illuminates a scene.

M

macro. In IBM Data Explorer, a sequence of modules that are be collected together. A macro cannot have recursion at any level.

makefile. A file used by the UNIX** utility “make” that contains a list of source files and commands required for maintaining current versions of Data Explorer programs. The modification times of the files determine what files have changed since the last time “make” was run, and therefore, what commands must be executed.

mdf. See module description file.

member. An individual unit in a group. A collection of members makes a group.

mesh Array. A compact Array that encodes multidimensional regularity of connections. It is a product of connection Arrays. The product is a set of interpolation elements in which the product has one element for each pair of elements in the two multiplicands, and the number of sample points in each element is the product of the number of sample points in each of the multiplicands' elements.

module. A primitive function that is provided as part of IBM Data Explorer, such as the Isosurface module. The same term is used for the function's icon, which is its visible representation in the VPE window.

Module Builder. A graphical user interface to assist in the creation of user-defined modules.

module description file (mdf). A module description file is used by a programmer who is adding a module to Data Explorer to describe information about the module that is needed by the system.

A module description file contains the name of the module, a short description of it, a category for the user interface to put the module in, and the names and descriptions of the input and output parameters. The module description file is used by the executive and the user interface to name parameters. The module description file is also used by the graphical user interface to form a tool icon in the proper category with the right number of input and output tabs.

MultiGrid. A MultiGrid Group is another kind of Group that is treated as a single entity. Like a Composite Field, all of the members of a MultiGrid Group must have the same type of data and the same type of connections. It is *not* required that members be disjoint and abutting however. Overlapping grids can be used along with the "invalid positions" and "invalid connections" components to define which grid is valid in a particular region.

N

netCDF. See Network Common Data Form.

Network Common Data Form (netCDF). A data abstraction that stores and retrieves scientific data structures in self-describing, multiple dimensional blocks. netCDF is accessible with C or FORTRAN. netCDF is not a database management system.

O

Object. A region of global memory that contains an identification of its type plus additional type-dependent information.

P

parallelism. The concurrent execution of modules.

partition. A member of a Composite Field.

partitioned Field. A Composite Field.

path array. A compact Array that encodes linear regularity of connections. It is a set of $n - 1$ line segments, where the ith line segment joins points i and $i + 1$.

pick. One Object that was picked using the mouse.

pick path. A list of integers that describe how to traverse an Object's hierarchy to reach a particular picked object.

pick structure. A Field that identifies objects picked using the mouse. It contains the components "positions," "pick paths," "picks," and "pokes."

poke. One screen location that was poked, or selected, by the user. One poke may result in none, one, or more than one pick.

positions. A component of a Field, consisting of a set of points.

private Object. Objects containing arbitrarily stored data. These Objects can be stored in the cache for later use.

product Array. A compact Array that encodes multidimensional positional regularity. It is the set of points obtained by summing one point from each of the terms in all possible combinations.

pseudo key. Accesses Objects in a hash table.

R

realization. The representation of data in terms of boundaries, surfaces, transparency, color, and other graphical, image, and geometric characteristics.

regular Array. A compact Array that is a set of n points lying on a line with constant spacing between them, representing 1-dimensional regular positions.

rendering. The generation of an image from some representation of an object. Rendering can be done from a surface representation, or from volumetric information.

ribbon. Ribbons are derived from lines (e.g., streamlines and streaklines). Ribbons may twist to indicate vorticity.

S

scripting language. The IBM Data Explorer command-line language. The scripting language is available for users who choose to write visual programs in a programming language. Data Explorer uses the scripting language to manage the execution of modules and to invoke visualization functions.

Series. A Series is intended to represent a single Field sampled across some parameter, such as time or temperature (e.g., a simulation of a CMOS device across a temperature range. Members of a Series have a position. A copy of the position is found in the "Series position" attribute.

Every member of the Series must have the same dimensionality, say *n*, the same data type, and the same connections element type. Members are stored in and retrieved from a Series Group by index rather than by name. Members cannot be retrieved by Series value.

shading. Applying lights to a surface during the rendering process.

shrink. Generally necessary only in a parallel environment; deletes the information, previously added by Grow, from neighboring partitions of a Composite Field. Composite Field members are returned to their original size.

stand-in. See *interactor stand-in*.

streaklines. Lines, sometimes called rakes, that show the path of particles over time.

string Object. A character string stored in an Object.

T

tiling. Combining shaded surface and volume interpolation elements to produce an image.

tool. A general term used to describe any of the icons used by IBM Data Explorer to build a visual program, specifically modules, macros, or interactor stand-ins.

tube. Tubes are derived from lines (e.g., streamlines and streaklines). Tubes may twist to indicate vorticity.

V

value. (1) A specific occurrence of an attribute (e.g. for the attribute "color"). (2) A quantity assigned to a constant, a variable, a parameter, or a symbol.

visual program. A user-specified interconnection of Data Explorer modules that performs a sequence of operations on data.

Visual Program Editor. Data Explorer graphical user interface for creating and editing visual programs and macros. See also *canvas*.

VPE. See Visual Program Editor.

volume. An object that fills a 3-dimensional space. A solid ball is a volume. A surface, however, is infinitely thin and does not fill space even though it exists in three dimensions.

volume rendering. A technique for creating an image that shows not just the surface of a 3-dimensional object, but its contents as well. The interior details being visualized may be physical, such as bone and muscle in a human body or the structure of a machine part, or they may be other characteristics such as fluid flow, heat, or stress.

Index

Glossary

Readers' Comments — We'd Like to Hear from You

IBM Visualization Data Explorer
Programmer’s Reference
Version 3 Release 1 Modification 4

Publication No. SC38-0497-06

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name	Address
Company or Organization	
Phone No.	

Readers' Comments — We'd Like to Hear from You
SC38-0497-06

IBM®

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Thomas J. Watson Research Center/Hawthorne
Data Explorer Development
P.O. Box 704
YORKTOWN HEIGHTS, NY
USA 10598-0704

Fold and Tape **Please do not staple** Fold and Tape

SC38-0497-06

Cut or Fold
Along Line

Printed in U.S.A.

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber

SC38-0497-06