



What's New in IDL 5.3



IDL Version 5.3
September, 1999 Edition
Copyright © Research Systems, Inc.
All Rights Reserved

Restricted Rights Notice

The IDL[®] software program and the accompanying procedures, functions, and documentation described herein are sold under license agreement. Their use, duplication, and disclosure are subject to the restrictions stated in the license agreement. Research Systems, Inc., reserves the right to make changes to this document at any time and without notice.

Limitation of Warranty

Research Systems, Inc. makes no warranties, either express or implied, as to any matter not expressly set forth in the license agreement, including without limitation the condition of the software, merchantability, or fitness for any particular purpose.

Research Systems, Inc. shall not be liable for any direct, consequential, or other damages suffered by the Licensee or any others resulting from use of the IDL software package or its documentation.

Permission to Reproduce this Manual

If you are a licensed user of this product, Research Systems, Inc. grants you a limited, nontransferable license to reproduce this particular document provided such copies are for your use only and are not sold or distributed to third parties. All such copies must contain the title page and this notice page in their entirety.

Acknowledgments

IDL[®] is a trademark of Research Systems Inc., registered in the United States Patent and Trademark Office, for the computer program described herein.

Numerical Recipes[™] is a trademark of Numerical Recipes Software. Numerical Recipes routines are used by permission.

GRG2[™] is a trademark of Windward Technologies, Inc. The GRG2 software for nonlinear optimization is used by permission.

NCSA Hierarchical Data Format (HDF) Software Library and Utilities
Copyright 1988-1998 The Board of Trustees of the University of Illinois
All rights reserved.

Portions of this software are copyrighted by INTERSOLV, Inc., 1991-1998.

Other trademarks and registered trademarks are the property of the respective trademark holders.



Research Systems, Inc. documentation is printed on recycled paper. Our paper has a minimum 20% post-consumer waste content and meets all EPA guidelines.



Contents

Chapter 1:	
Overview of New Features in IDL 5.3	9
Visualization and Analysis Enhancements	10
IDL Language Enhancements	21
File I/O Enhancements	24
Development Environment Enhancements	27
Scientific Data Formats Enhancements	31
IDL GUIBuilder Enhancements	33
IDL ActiveX Control Enhancements	34
Installation and Licensing Enhancements	39
IDL DataMiner Enhancements	43
Documentation Enhancements	44
New Object Features	49

New Routines	58
New and Updated Keywords/Arguments	65
New Environment Variables	80
Routines Obsoleted in IDL 5.3	81
Platforms Supported in this Release	82
Chapter 2:	
Creating IDL Projects	83
Overview	84
Where to Store Source Files for a Project	86
Creating a Project	87
Opening, Closing, and Saving Projects	89
Adding, Moving, and Removing Files	90
Working with Files in a Project	93
Setting the Options for a Project	96
Selecting the Build Order	99
Compiling an Application from a Project	101
Building a Project	102
Running an Application from a Project	104
Exporting a Project	105
About IDL Developer's Kit Licenses	107
Chapter 3:	
IDL Development Environment Enhancements	109
Enhanced Breakpoint Functionality	110
New IDL Functions and Procedures Context Menu for Windows and Motif ...	114
New Color/Font Style Coding for Source Files on UNIX	115
Enhanced IDL Macros Support	117
Chapter 4:	
IDL Macros for Importing Data	119
Overview	120
Using Macros to Import Image Files	121

Using Macros to Import ASCII Files	125
Using Macros to Import Binary Files	131
Using Macros to Import HDF Files	137
Chapter 5:	
New IDL Routines	141
.FULL_RESET_SESSION	142
.RESET_SESSION	143
ADAPT_HIST_EQUAL	145
BINARY_TEMPLATE	147
CDF_COMPRESSION	149
COMPILE_OPT	153
CW_FILESEL	156
CW_LIGHT_EDITOR	158
CW_LIGHT_EDITOR_GET	162
CW_LIGHT_EDITOR_SET	164
CW_PALETTE_EDITOR	166
CW_PALETTE_EDITOR_GET	172
CW_PALETTE_EDITOR_SET	173
DIALOG_READ_IMAGE	174
DIALOG_WRITE_IMAGE	176
DLM_LOAD	178
DRAW_ROI	179
ENABLE_SYSRTN	181
EOS_GD_QUERY	183
EOS_PT_QUERY	185
EOS_QUERY	186
EOS_SW_QUERY	187
GET_DRIVE_LIST	189
GRID_TPS	190
IMAGE_STATISTICS	193
ISOCONTOUR	196

ISOSURFACE	199
LOCALE_GET	201
MESH_CLIP	202
MESH_DECIMATE	204
MESH_ISSOLID	206
MESH_MERGE	207
MESH_NUMTRIANGLES	209
MESH_SMOOTH	210
MESH_SURFACEAREA	212
MESH_VALIDATE	214
MESH_VOLUME	216
MORPH_CLOSE	217
MORPH_DISTANCE	219
MORPH_GRADIENT	222
MORPH_HITORMISS	224
MORPH_OPEN	226
MORPH_THIN	228
MORPH_TOPHAT	229
MSG_CAT_CLOSE	231
MSG_CAT_COMPILE	232
MSG_CAT_OPEN	234
PARTICLE_TRACE	236
QUERY_IMAGE	239
QUERY_WAV	242
READ_BINARY	243
READ_IMAGE	245
READ_WAV	247
STRCMP	248
STREAMLINE	250
STREGEX	252
STRJOIN	256
STRMATCH	257

STRSPLIT	260
STRUCT_HIDE	264
TETRA_CLIP	266
TETRA_SURFACE	268
TETRA_VOLUME	269
VALUE_LOCATE	271
VECTOR_FIELD	273
WATERSHED	275
WRITE_IMAGE	277
WRITE_WAV	278
XOBJVIEW	279
Chapter 6:	
New Objects	283
IDLanROI	284
IDLanROIGroup	307
IDLffLanguageCat	324
IDLgrBuffer::GetDeviceInfo	328
IDLgrClipboard::GetDeviceInfo	330
IDLgrROI	332
IDLgrROIGroup	342
IDLgrVRML::GetDeviceInfo	347
IDLgrWindow::GetDeviceInfo	349
Index	351



Chapter 1: Overview of New Features in IDL 5.3

This chapter contains the following topics:

Visualization and Analysis Enhancements	10	IDL DataMiner Enhancements	43
IDL Language Enhancements	21	Documentation Enhancements	44
File I/O Enhancements	24	New Object Features	49
Development Environment Enhancements	27	New Routines	58
Scientific Data Formats Enhancements	31	New and Updated Keywords/Arguments	65
IDL GUIBuilder Enhancements	33	New Environment Variables	80
IDL ActiveX Control Enhancements	34	Routines Obsolete in IDL 5.3	81
Installation and Licensing Enhancements	39	Platforms Supported in this Release	82

Visualization and Analysis Enhancements

The following enhancements have been made in the area of Visualization and Analysis in the IDL 5.3 release:

- [Image Processing Improvements](#)
- [3D Visualization Improvements](#)
- [New Vector Output of Object Graphics](#)
- [New Sub-Rectangle Support for Image Display](#)
- [Enhanced User Control Over Axis Label Orientation](#)
- [Enhanced Query Support for Objects Graphics Devices](#)
- [Enhanced Sparse Matrix Functionality](#)
- [New Object Viewer](#)

Image Processing Improvements

Additional image processing tools are included in the IDL 5.3 release. The new functionality is designed to increase IDL's capabilities in quantitative image analyses, such as those needed to analyze images from medical scanning technologies, satellite data, microscopes, telescopes, etc.

New Routines, Objects, and Compound Widgets

The following list describes the new functionality added in this release and the type of image processing technique where it is used.

- **[ADAPT_HIST_EQUAL](#)** — Performs adaptive histogram equalization, a form of automatic image contrast enhancement. This method of automatic contrast enhancement has proven to be broadly applicable to a wide range of images and to have demonstrated effectiveness.
- **[GRID_TPS](#)** — A geometric manipulation and interpolation technique which uses thin plate splines to interpolate a set of values.
- **[IMAGE_STATISTICS](#)** — Generates sample statistics for an array of values.
- **[CW_PALETTE_EDITOR](#)** — Creates a compound widget to display and edit color palettes. This compound widget facilitates displaying and editing of color palettes used in image processing.

A set of additional morphological functions are now available in IDL for use in image processing. The following list shows the morphological routines included in the IDL 5.3 release with a short description of the new functionality.

- **MORPH_CLOSE** — Applies the closing operator to a binary or grayscale image. It is simply a dilation operation followed by an erosion operation. The result of a closing operation is that small holes and gaps within the image are filled, yet the original sizes of the primary foreground features are maintained.
- **MORPH_DISTANCE** — Estimates N -dimensional distance maps, which contain for each foreground pixel the distance to the nearest background pixel using a given norm. The distance map is useful for a variety of morphological operations: thinning, erosion and dilation by discs of radius “ r ”, and granulometry.
- **MORPH_GRADIENT** — Applies the morphological gradient operator to a grayscale image. The practical result of a morphological gradient operation is that the boundaries of features are highlighted.
- **MORPH_HITORMISS** — Applies the hit-or-miss operator to a binary image.
- **MORPH_OPEN** — Applies the opening operator to a binary or grayscale image. The result of an opening operation is that small features (e.g., noise) within the image are removed, yet the original sizes of the primary foreground features are maintained.
- **MORPH_THIN** — Implements a thinning operator on binary images.
- **MORPH_TOPHAT** — Applies the top-hat operator to a grayscale image. Applying the top-hat operator shows the bright peaks within the image.
- **WATERSHED** — Applies watershed segmentation to a binary image.

Four object classes have also been added for graphical and analysis capabilities on regions of interest (ROIs). These are described in the “[New Object Features](#)” section of this chapter.

- **IDLanROI** — The IDLanROI object class provides an analytical representation of a region of interest.
- **IDLanROIGroup** — Analytical representation of a group of regions of interest.
- **IDLgrROI** — An Object Graphics representation of a region of interest.

- **IDLgrROIGroup** — Object Graphics representation of a group of regions of interest.

Changes to Existing Image Processing Routines

New keywords have been added to pre-existing image processing routines:

- **ERODE** and **DILATE** — Added `UINT`, `ULONG`, `PRESERVE_TYPE`, and `THRESHOLD` keywords. These keyword expand the data type support for the functions to include the `UINT` and `ULONG` data types in addition to the existing byte type, and also allows the output from `LABEL_REGION` to be used as input to the `ERODE` and `DILATE` functions.
- **DILATE** — Two additional keywords, `BACKGROUND` and `CONSTRAINED` have been added for constrained dilation support.
- **LABEL_REGION** — The argument for this function has been changed from *Image* to *Data*, and now allows *n*-dimensional arrays to be labeled.

Two new keywords (`ALL_NEIGHBORS` and `ULONG`) have been added and one keyword (`EIGHT`) has been obsoleted. The `ALL_NEIGHBORS` keyword functionally replaces the obsoleted keyword, and `ULONG` allows the output array to be an unsigned long integer instead of a short.

- **TOTAL** — The `CUMULATIVE` keyword has been added.

Example Program Using New Image Processing Tools

A new example is included with IDL 5.3 that show how to use the ROI improvements. This example (`roi_example.pro`) is located in the `examples/objects` directory and shows how you can easily create applications

using the new `IDLanROI` and `IDLgrROI` objects. You can run these examples by simply typing `roi_example` at the IDL command line.

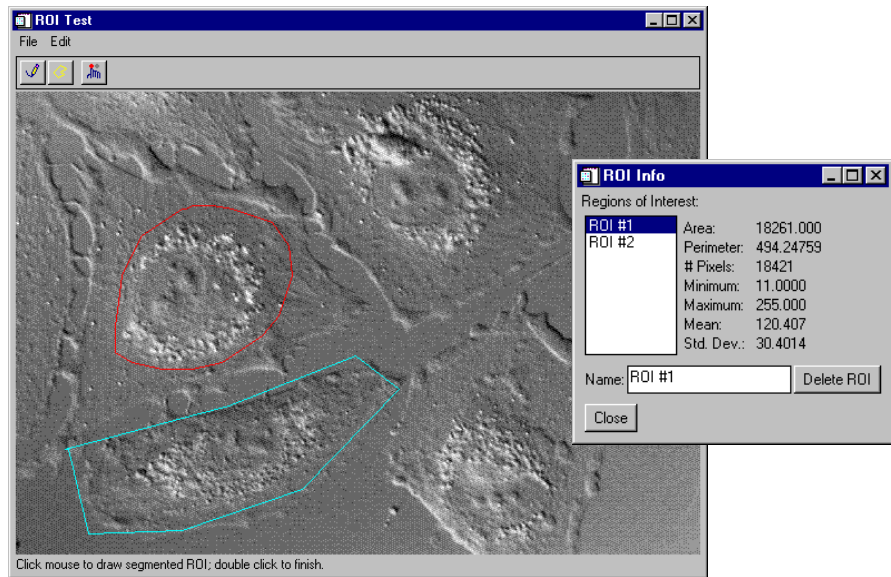


Figure 1-1: Example of New Image Processing Tools (`roi_example.pro`)

3D Visualization Improvements

Additional 3D visualization functionality is included in the IDL 5.3 release. These functions provide a suite of focused tools in the form of compound widgets in addition to some new and consistent end-user routines.

Three new routines have been added for feature extraction functionality:

- **ISOCONTOUR** — Exposes the contouring algorithm found in the `IsoContour` object.
- **ISOSURFACE** — Expands on the existing `SHADE_VOLUME` algorithm.

The following routines have been added for polygon mesh functionality:

- **MESH_CLIP** — Clips a polygonal mesh to an arbitrary plane in space and return a polygonal mesh of the remaining portion.

- **MESH_DECIMATE** — Deletes points in a polygonal mesh that satisfy a coplanar or co-linear condition and replaces the resulting hole with a new triangulation.
- **MESH_ISSOLID** — Computes various mesh properties and enables IDL to determine if a mesh is a solid. If the mesh can be considered a solid, routines can compute the volume of the mesh.
- **MESH_MERGE** — Merges two polygonal meshes.
- **MESH_NUMTRIANGLES** — Computes various mesh properties and enables IDL to determine the number of triangles in the mesh.
- **MESH_SMOOTH** — Performs spatial smoothing on a polygon mesh.
- **MESH_SURFACEAREA** — Computes various mesh properties and enables IDL to determine the mesh surface area, including integration of other properties interpolated on the surface of the mesh.
- **MESH_VALIDATE** — Checks for NaN values in vertices and removes unused vertices.
- **MESH_VOLUME** — Computes various mesh properties and enables IDL to determine the volume that the mesh encloses.

Three new routines have been added for tetrahedral mesh functionality:

- **TETRA_CLIP** — Clips a tetrahedron mesh to an arbitrary plane in space and returns a tetrahedral mesh of the remaining portion.
- **TETRA_SURFACE** — Extracts a polygon mesh as the exterior surface of a tetrahedral mesh.
- **TETRA_VOLUME** — Computes properties of a tetrahedral mesh array.

The following routines have been added for field visualization functionality:

- **PARTICLE_TRACE** — Traces the path of a mass-less particle through a vector field and allows the user to specify a set of starting points and a vector field.
- **STREAMLINE** — Computes a line that traces the path of a particle through a constant vector field.
- **VECTOR_FIELD** — Used to place colored, orientated vectors of specified length at each vertex in an input vertex array.

The following new compound widgets have been added for graphics functionality:

- **CW_LIGHT_EDITOR** — Creates a compound widget to edit properties of existing IDLgrLight objects in a view.
- **CW_PALETTE_EDITOR** — Creates a compound widget to display and edit color palettes. This compound widget facilitates displaying and editing of color palettes used in image processing.

The following routine has been enhanced in the area of visualization through the addition of new keywords and arguments.

- **EXTRACT_SLICE** — New keywords and arguments have been added to support anisotropy, alternative forms of defining the plane for slicing, and allows for a vertex grid to be generated without sampling the data.

Example Programs Using New 3D Visualization Tools

Two new examples are included with IDL 5.3 that show how to use the 3D Visualization improvements. These examples are:

- `decimate.pro` — Shows the use of the new MESH_ routines, in particular, the MESH_DECIMATE routine
- `tetra.pro` — Shows the use of the TETRA_ routines, in particular the TETRA_CLIP routine

The source for these examples are included with IDL in the `examples/objects` directory and show how you can easily create applications using the new features in

IDL 5.3. You can run these examples by simply typing either `decimate` or `tetra` at the IDL command line.

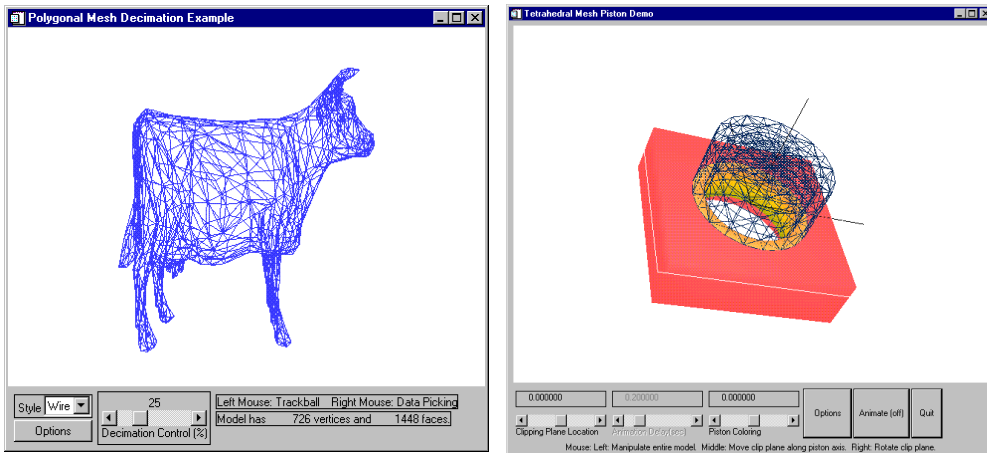


Figure 1-2: Examples of New 3D Visualization Tools (*decimator.pro* on the left, *tetra.pro* on the right)

New Vector Output of Object Graphics

IDL now includes support for vector output for both the clipboard and printer. This is to improve performance of printing as well as moving graphics from IDL into other applications.

New keywords have been added to support the new vector output to the `IDLgrClipboard` and `IDLgrPrinter` object classes.

- **IDLgrClipboard::Draw**
 - **FILE** — Use this keyword to write the output to a file instead of the clipboard.
 - **POSTSCRIPT** — Use this keyword to specify that the generated output should be in PostScript format.
 - **VECTOR** — Use this keyword to generate the graphics primitives in either bitmap or vector format.

- **IDLgrPrinter::Draw**
 - **VECTOR** — Use this keyword to generate the graphics primitives in either bitmap or vector format.

For more information, see “[New and Updated Keywords to IDL Object Methods](#)” on page 53.

Vector Output vs. Bitmap Output

Using the vector output does not always ensure a better result. There are trade-offs involved depending on the type of graphics editing you are using, and the intended format for the final result.

When to use Vector Output for the Printer

Vector output to the printer results in faster print times for relatively simple scenes. An increase in the scene complexity results in slower print times due to the larger file being printed. For complex scenes, bitmap output may be faster. But for simple scenes that contain only simpler line plots, the vector format will print much faster and generate an identical image.

When to use Bitmap Output for the Printer

Bitmap output should always be used whenever you need to preserve image attributes of complex 3D scenes. Again, this includes lighting and shading effects, precise depth buffering, and other advanced graphical effects.

When to use Vector Output for the Clipboard

Vector output works very well for editing individual objects in a scene using a graphical object editor like the one in Microsoft Word; however, if you’re editing an image with a bitmap editor, such as Microsoft Paint, vector output is not recommended.

Use vector output for accurately resizing a pasted clipboard object. If the pasted clipboard object is in vector format, resizing produces good results. This feature is extremely useful for producing images to include in documents, when resizing is often necessary to make the image fit in the document’s confined spaces. Although bitmaps can also be resized, the quality of the image degrades as the bitmap is enlarged or reduced in size.

Vector output provides efficiency gains by reducing the memory required for a clipboard object, depending on the scene content. Simple images with large dimensions require a lot of space in the clipboard.

When to use Bitmap Output for the Clipboard

Bitmap output to the clipboard is preferable when trying to preserve all image attributes for complex 3D scenes. Image attributes you may wish to preserve using bitmap output include lighting and shading effects, precise depth buffering, and other advanced graphical effects. Many of these special effects are lost when a vector representation is used instead.

New Sub-Rectangle Support for Image Display

You can now access, process, and display sub-rectangles of a given data set for imaging by using the new `SUB_RECT` keyword for the `IDLgrImage::Init` method. This is useful when you have a very large image but are only interested in viewing a sub section of the image. This keyword specifies the position of the lower left-hand corner and the dimensions of the sub-rectangle to display.

For more information, see [“New and Updated Keywords to IDL Object Methods”](#) on page 53.

Enhanced User Control Over Axis Label Orientation

The following new keywords have been added to the `IDLgrAxis::Init` object method to allow the user to specify the orientation of the text items used to label tickmarks on the `IDLgrAxis` object.

- **TEXTALIGNMENTS** — This keyword specifies the horizontal and vertical justification of the tick text: left- or right-, and top- or bottom-justified.
- **TEXTBASELINE** — This keyword describes the direction in which the baseline of the tick text is to be oriented.
- **TEXTUPDIR** — This keyword describes the direction in which the up-vector of the tick text is to be oriented.

For more information, see [“New and Updated Keywords to IDL Object Methods”](#) on page 53.

Enhanced Query Support for Objects Graphics Devices

The following items of information are now available for query in Object Graphics:

- OpenGL renderer description string
- Maximum view port dimensions
- Maximum texture dimensions

- Approximate performance measurement (e.g., number of polygons per second)

You can obtain this information and use it to optimize your Object Graphics code by understanding the limitations of a device.

The following methods have been added:

- [IDLgrBuffer::GetDeviceInfo](#)
- [IDLgrClipboard::GetDeviceInfo](#)
- [IDLgrVRML::GetDeviceInfo](#)
- [IDLgrWindow::GetDeviceInfo](#)

For more information, see [Chapter 6, “New Objects”](#).

Enhanced Sparse Matrix Functionality

In previous versions of IDL, there was no way to create a sparse matrix without first creating a full storage matrix. The [SPRSIN](#) function has been enhanced to allow conversion of a list of subscripts and values to row-indexed sparse storage mode. This is a more efficient method than converting an array when the density of the matrix is low. For more information, see [“New and Updated Keywords/Arguments”](#) on page 65.

New Object Viewer

The new [XOBJVIEW](#) procedure allows you to quickly and easily view and manipulate IDL Object Graphics on screen. This procedure displays a widget containing buttons that allow you to rotate, pan, and scale the object using your mouse.

See [Chapter 5, “New IDL Routines”](#) for complete documentation on the XOBJVIEW procedure.

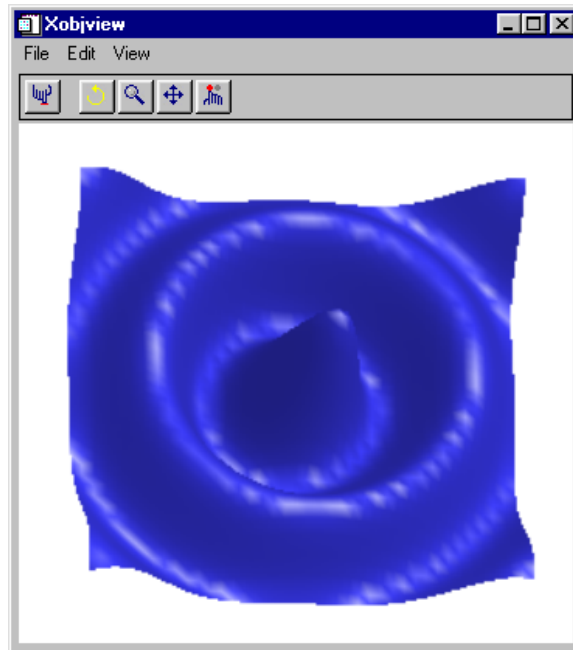


Figure 1-3: The XOBJVIEW draw widget

IDL Language Enhancements

The following enhancements have been made in the IDL language in the IDL 5.3 release:

- [Internationalization Support for IDL Applications](#)
- [New String Processing Functionality](#)
- [New IDL Session Reset Commands](#)
- [New COMPILE_OPT Statement](#)
- [New Output Options for the HELP Procedure](#)
- [Raised Limits](#)

Internationalization Support for IDL Applications

New support for internationalization provides you with the ability to create a message catalog that you can call from within your application. This catalog can then be translated so that you can support multiple languages. Several new procedures and an object class have been added to support internationalizing your IDL application.

Three new procedures have been added. They are:

- [MSG_CAT_COMPILE](#) — Creates an IDL language catalog file.
- [MSG_CAT_CLOSE](#) — Closes an IDL language catalog file.
- [MSG_CAT_OPEN](#) — Returns the specified object from an IDL language catalog file.

For more information, see [Chapter 5, “New IDL Routines”](#).

A new `IDLffLanguageCat` object class has been added that has the following methods:

- [IDLffLanguageCat::IsValid](#) — Determines whether the `IDLffLanguage Cat` object has a valid catalog.
- [IDLffLanguageCat::Query](#) — Returns the language string associated with a specified key.
- [IDLffLanguageCat::SetCatalog](#) — Specifies a catalog to use.

For more information, see [Chapter 6, “New Objects”](#).

New String Processing Functionality

The following new string processing routines have been added. See [Chapter 5, “New IDL Routines”](#) for complete documentation on these new routines:

- **STRCMP** — Compares two strings. Can perform case-insensitive comparison of first N characters more easily than using the EQ operator.
- **STREGEX** — Performs regular expression matching. Regular expressions are a very powerful way to match arbitrary text. Regular expressions are an integral part of many UNIX tools, including awk, egrep, lex, perl, and sed, as well as many text editors. Regular expressions are slower than simple pattern matching algorithms, but are vastly more powerful than simple pattern matching, and can easily handle tasks that would be difficult or impossible otherwise.
- **STRJOIN** — Collapses a string scalar or array into merged strings. The separator string used between the joined strings can be specified.
- **STRMATCH** — Compares its search string, which can contain wild card characters, against the input string expression.
- **STRSPLIT** — Splits its input string argument into separate sub-strings, according to the specified pattern.

The following existing string routines have been enhanced:

- **STRMID**
 - The *First_Character* and *Length* arguments can now be arrays.
 - REVERSE_OFFSET — This new keyword specifies that *First_Character* should be counted from the end of the string backwards. This allows simple extraction of strings from the end.
- **STRPOS**
 - REVERSE_OFFSET — Normally, the value of the *Pos* argument is used as an offset from the beginning of the expression towards the end. Set this keyword to use it as an offset from the last character of the string moving towards the beginning. This keyword makes it easy to position the starting point of the search at a fixed offset from the end of the string.
 - REVERSE_SEARCH — STRPOS usually starts at *Pos* and moves toward the end of the string looking for a match. If REVERSE_SEARCH is set, the search instead moves towards the beginning of the string. This keyword obsoletes the RSTRPOS function.

New IDL Session Reset Commands

The new executive commands `.RESET_SESSION` and `.FULL_RESET_SESSION` allow you to reset much of the state of an IDL session without having to exit and restart the IDL session. See [Chapter 5, “New IDL Routines”](#) for complete documentation on these new commands.

New `COMPILE_OPT` Statement

The new `COMPILE_OPT` statement allows you to provide the IDL compiler with information that changes some of the default rules for how to compile a function or procedure. See [Chapter 5, “New IDL Routines”](#) for complete documentation on this new statement.

New Output Options for the `HELP` Procedure

Several new keywords to the `HELP` procedure provide greater control over the information returned by the `HELP` procedure. The `BRIEF` keyword causes terse, summary style output, while the `FULL` keyword causes full, unfiltered output. The `FUNCTIONS` and `PROCEDURES` keywords allow you to limit output produced by the `ROUTINES` and `SOURCE_FILES` keywords to either functions or procedures.

Raised Limits

Limits for the following have been raised in IDL 5.3:

- The limit on the number of elements allowed in an array concatenation (e.g. [] operators) has been raised from 90 to 65535.
- The limit on the number of plain or keyword arguments to an IDL function or procedure has been raised from 64 to 65535.

File I/O Enhancements

The following enhancements have been made in the IDL language in the IDL 5.3 release:

- [New Support for GZIP File Compression/Decompression](#)
- [New File Input/Output/Query Functionality](#)
- [New Support for .WAV Audio Files](#)
- [Enhanced Support for Tiff Images](#)
- [Improved Macros for Importing Data](#)

New Support for GZIP File Compression/Decompression

IDL 5.3 now has support for GZIP file compression/decompression. You can now read and write data files that use GZIP as well as create .SAV files which can be compressed as well. This allows you to read and write data files as well as create .SAV files that have been compressed which greatly reduces the disk space required.

The [OPEN](#) procedures (OPENR, OPENW, and OPENU) now have a COMPRESS keyword that allows you to read and write all data to the file in the standard GZIP format. This means that IDL's compressed files are 100% compatible with the widely available gzip and gunzip programs.

The [SAVE](#) procedure also has a new COMPRESS keyword that causes IDL to write all data to the SAVE file using the ZLIB compression library to reduce its size.

IDL Demo files now use this feature so they are compressed, resulting in a smaller IDL installation size.

New File Input/Output/Query Functionality

Additional file input/output functionality is included in the IDL 5.3 release. This allows you to easily read ASCII, binary, image, and audio files into IDL. The following list describes the areas of new functionality:

- The following routines have been added for file location functionality:
 - [GET_DRIVE_LIST](#) (Windows and Mac only) — Returns a string array of the names of valid drives / volumes for the file system.

- The following routines have been added for reading binary data:
 - **BINARY_TEMPLATE** — Allows the user to interactively generate a template structure for a binary file. You can then use this template to read in binary file with the READ_BINARY function.
 - **READ_BINARY** — Reads the contents of a binary file using a passed template or basic command line keywords.
- The following routines have been added for general image functionality:
 - **QUERY_IMAGE** — Reads the header of a file and determines if it is recognized as an image file.
 - **READ_IMAGE** — Reads the image contents of a file and returns the image in an IDL variable.
 - **WRITE_IMAGE** — Writes an image and its color table vectors, if any, to a file of a specified type.
- The following compound widgets and dialogs have been added for compound widgets and dialogs functionality:
 - **CW_FILESEL** — A compound widget for file selection.
 - **DIALOG_READ_IMAGE** — A graphical user interface used for reading image files.
 - **DIALOG_WRITE_IMAGE** — A graphical user interface used for writing image files.

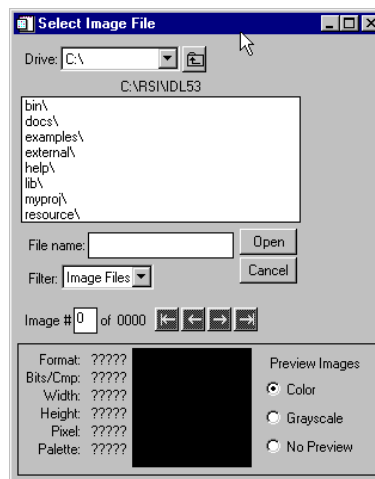


Figure 1-4: DIALOG_READ_IMAGE

New Support for .WAV Audio Files

These new routines add support for input and output of .WAV audio files.

- **QUERY_WAV** — Checks that the file is actually a .WAV file and that the READ_WAV function can read the data in the file.
- **READ_WAV** — Reads the audio stream from the named .WAV file.
- **WRITE_WAV** — Writes the audio stream from the named .WAV file.

Enhanced Support for Tiff Images

IDL now supports single or multi-channel TIFF images. The following routines have been enhanced:

- **READ_TIFF** — This procedure now reads single and multi-channel images and returns the image and color table vectors.
- **WRITE_TIFF** — This procedure can now write TIFF files with one or more channels where each channel can contain 8, 16, 32, or floating point pixels.

Improved Macros for Importing Data

Items have been added to the IDL Development Environment Macros menu and the Tool Bar to make importing of image, ASCII, binary, and HDF data into IDL even easier by giving you dialogs that step you through the process. For more information, see [Chapter 4, “IDL Macros for Importing Data”](#).

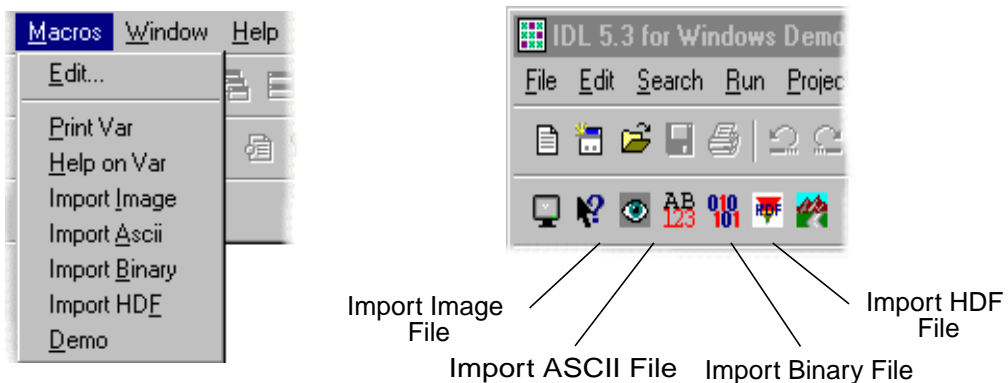


Figure 1-5: Importing Data Macros Menu (Left) and Tool Bar Buttons (Right)

Development Environment Enhancements

The following enhancements have been made in the IDL language in the IDL 5.3 release:

- [New IDL Projects](#)
- [Enhanced Breakpoint Functionality](#)
- [New IDL Functions/Procedures Context Menu](#)
- [New Color/Font Style Coding for Source Files on Motif](#)
- [Enhanced IDL MACRO Support](#)

New IDL Projects

IDL Projects allow you to easily develop applications in IDL. You can manage, compile, run, and create distributions of all the files you will need to develop your IDL application. All of your application files can be organized so that they are easier to access and easier to export to other developers, colleagues, or users. IDL Projects are a great benefit to development teams working on a large project as well as individual developers managing multiple projects.

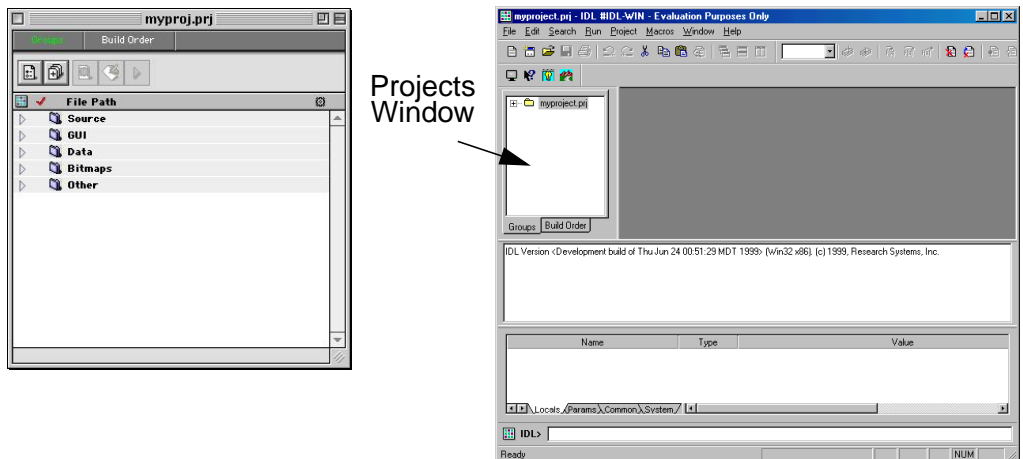


Figure 1-6: Projects Window for Macintosh (left) and Windows (right)

Access to all Files in Your Application

IDL Projects have an easy to use interface for grouping:

- IDL source code files (.pro)
- GUI files (.prc) created with IDL GUIBuilder
- Data files (ASCII text or binary)
- Image files (.tif, .gif, .bmp, etc.)
- Other files (help files, .sav files, etc.)

After you add all of your files to your project, you can simply double click on .pro files to open them in the IDL editor or .prc files to open them in the IDL GUIBuilder.

Working with Files in Your Project

IDL projects make it easy to add, remove, move, edit, compile, and test files in your project.

All of your workspace information is saved as well. If you save and exit your project with open files, when you open your project, those same files will be opened automatically for you.

IDL projects also store and retain breakpoint information. There is no need to reset breakpoints every time you open the project.

Compiling and Running Your Application

Compiling and running applications is fast and easy. You can compile all of your source files or just the files that you have modified and then run your application through the Projects menu. You can customize how your application is compiled and run by specifying options for your project.

Creating IDL Runtime Distributions

Once you have completed your application, you can quickly and easily create an IDL Runtime distribution with a new automated process. If you have purchased the IDL Developer's Kit, your application is automatically licensed for distribution.

Exporting Your Applications

You can easily move your application to another platform or distribute your source code to colleagues by exporting your project. All your source code, GUI files, data files, and image files are copied to a directory you specify.

For more information on IDL Projects and Developer's Kit, see [Chapter 2, "Creating IDL Projects"](#).

Enhanced Breakpoint Functionality

Breakpoints have been enhanced in IDL 5.3. You can now selectively create, delete, enable, disable, and set other options for breakpoints from one dialog. Other features include new keywords to the BREAKPOINT procedure as well as the ability to add breakpoints to a file that has not been compiled. For more information, see [Chapter 3, "IDL Development Environment Enhancements"](#).

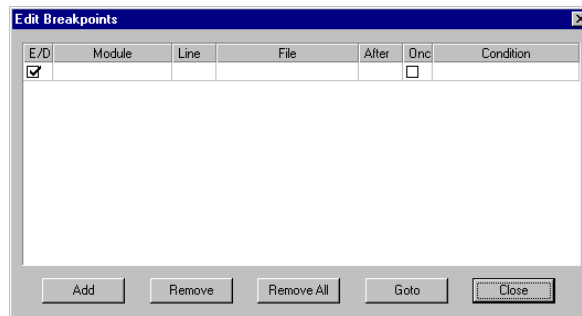


Figure 1-7: New Edit Breakpoints Dialog

New IDL Functions/Procedures Context Menu

Previously only available on the Macintosh, the IDL Function/Procedure Context Menu has been added to the Windows and Motif versions of IDL. The IDL Function/Procedure Context Menu allows you to navigate between the different procedures and functions you have defined in the current file you have open in the IDL Editor. For more information, see [Chapter 3, "IDL Development Environment Enhancements"](#).

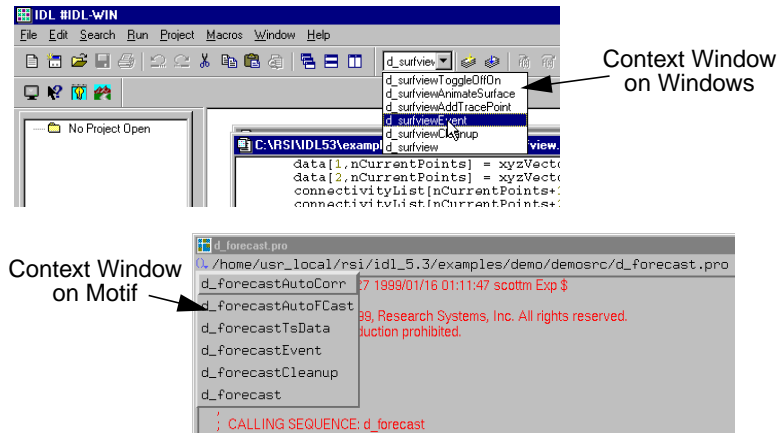


Figure 1-8: IDL Function/Procedure Context Menu on Windows (top) and Motif (bottom)

New Color/Font Style Coding for Source Files on Motif

Previously available on Windows and Macintosh platforms, color and font style coding has been added to IDL for Motif. This allows you to color code and specify different font styles for the different types of IDL statements that appear in the IDL Editor window. For more information, see [Chapter 3, “IDL Development Environment Enhancements”](#).

Enhanced IDL MACRO Support

New Command Stream Substitution

The %? substitution string has been added in IDL 5.3. This substitution string displays a dialog for a user to enter a value to pass to a macro. For more information, see [Chapter 3, “IDL Development Environment Enhancements”](#).

New Support for Command Stream Substitutions on Macintosh

Command stream substitutions are now available on the Macintosh. You can use command stream substitutions to include certain types of information into IDL Macros. For more information, see [Chapter 3, “IDL Development Environment Enhancements”](#).

Scientific Data Formats Enhancements

The following enhancements have been made in the IDL language in the IDL 5.3 release:

- [New Support for Compression of CDF and HDF SD Data Sets](#)
- [New HDF-EOS Query Routines](#)
- [New EOS_EXISTS Function](#)
- [HDF_BROWSER Enhancements](#)
- [Updated Library Versions](#)

New Support for Compression of CDF and HDF SD Data Sets

The new [CDF_COMPRESSION](#) procedure allows you to set or retrieve the compression mode of a CDF file and/or variable.

The new [HDF_SD_SETCOMPRESS](#) procedure compresses an existing HDF SD data set or sets the compression method of a newly created HDF SD data set.

New HDF-EOS Query Routines

The following new HDF-EOS query routines have been added:

- [EOS_GD_QUERY](#) — Returns information about a specified grid.
- [EOS_PT_QUERY](#) — Returns information about a specified point.
- [EOS_QUERY](#) — Returns information about the makeup of an HDF-EOS file.
- [EOS_SW_QUERY](#) — Returns information about a specified swath.

New EOS_EXISTS Function

The new [EOS_EXISTS](#) function allows you to determine whether the HDF-EOS extensions are supported on the current platform.

HDF_BROWSER Enhancements

The following new features have been added to the [HDF_BROWSER](#) function:

- [VData/VGroup Data Access](#) — This allows access to VData and VGroup data within HDF files.

- **New Show3 Preview Type** — This new preview type combines an image, a surface plot of the image data, and a contour plot of the images data in a single tri-level display.

Updated Library Versions

IDL now supports the following library versions:

- Common Data Format (CDF) 2.6r7
- Hierarchical Data Format (HDF) 4.1r3
- Hierarchical Data Format Earth Observing System (HDF-EOS) 2.4

IDL GUIBuilder Enhancements

The IDL GUIBuilder has been enhanced so that the event file is no longer overwritten when you generate it. When the event file is built, pre-existing functions and procedures in the event file are not overwritten. The IDL GUIBuilder now appends new event routines to the end of the file so that you can now more easily maintain your event file.

Because of this change, menu options that generate the event file and source code separately are no longer needed. There is now only one menu item:

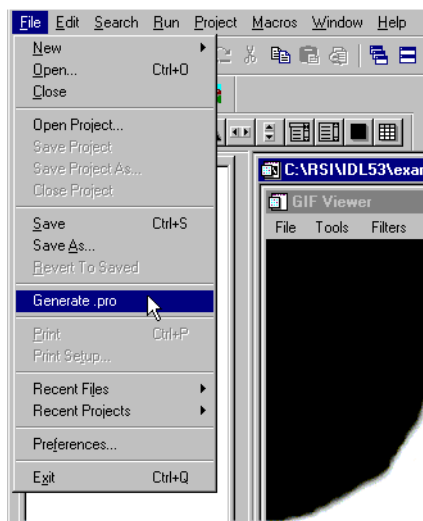


Figure 1-9: New Menu Configuration for IDL GUIBuilder

IDL ActiveX Control Enhancements

IDL 5.3 includes a new version of the IDLDrawX ActiveX control. The control is now named IDLDrawX2.

Why Was a New Version of the Control Created?

One of the features of COM is that interfaces are immutable. That is to say that when an interface is created you “contractually” agree that the interface won’t change. Changes require that a new interface (or version) be created. Since the IDL ActiveX control is a COM object it is bound by this agreement. Because we have made improvements to the ActiveX control interface by adding new methods and properties, it was necessary that we create a new ActiveX control with the new interface.

What Must You Change to Take Advantage of the Control?

If you are a Visual Basic user, you need to add the “IDLDrawX2 ActiveX Control Module” to your project and remove the “IDLDrawX ActiveX Control Module” from your project. The source code need not change.

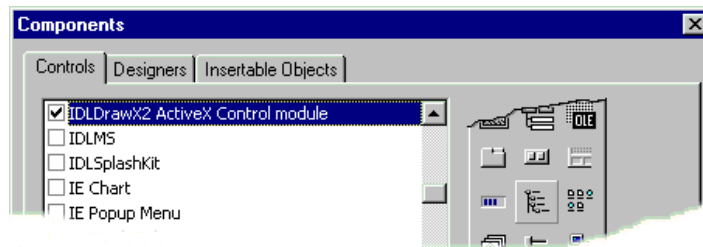


Figure 1-10: IDL DrawX2 ActiveX Control

What About the Previous ActiveX Control?

While version 1.0 of the IDLDrawX control will continue to work with new versions of IDL, it is no longer supported and will not be shipped with IDL. It is recommended that you upgrade to the new version to take advantage of new features and bug fixes.

Why Should You Upgrade?

The new control has a number of new features including printing support, dual interface control, and new memory improvements. The rest of this section details the improvements made in the new version of the IDL ActiveX control.

New Dual Interface Control

Starting in IDL 5.3, the IDLDrawX2 ActiveX control is a dual interface control. A dual interface control is an ActiveX control that can be bound to at both compile time through a vTable and at runtime through a dispatch interface. Scripting clients such as java script, VB Script, and VBA use runtime binding while compiled languages such as Visual C++ and some VB applications use compile time binding. This now gives compiled languages the ability to call methods on the IDLDrawX2 ActiveX control directly rather than through a dispatch interface.

New Printing Support

Applications that have printing and print preview capabilities can now take advantage of printing support within the new IDLDrawX2 ActiveX control. This new capability is transparent to applications that have built-in print capabilities. For applications that do not have built-in print capabilities, you can call the new Print method to get your output. The new enhancements that support this capability are:

- A new Print method
- A new BufferId property

Print

This method prints the contents of the ActiveX control to the current default printer for both Direct and Object Graphics windows. The Print method will print the contents of the window at screen resolution with a Direct Graphics window. For information about controlling print resolution of an object graphics window, see the [BufferId](#) property.

Parameters

XOffset: The X offset to print the graphic in 0.01 of a millimeter.

YOffset: The Y offset to print the graphic in 0.01 of a millimeter.

Width: The desired width of the printed graphic in 0.01 of a millimeter.

Height: The desired height of the printed graphic in 0.01 of a millimeter.

The X offset plus the width should be less than or equal to the width of a single page. The Y offset plus the height should be less than or equal to the height of a single page. The origin of the offset 0,0 is in the upper left corner of a page. If these values are set to 0, the ActiveX control will print a graphic in the upper left corner of the page with the size of the graphic approximating the size of the image on the screen.

Returns

BOOL: TRUE if printing succeeded.

BufferId

The BufferId controls the type of print output you receive when printing with an Object Graphics window (when the GraphicsLevel property is set to 2).

1. A value of -1 will cause the graphics to print using vector output. This format is suitable for line graphs and mesh surfaces.
2. A value of 0 will cause the graphics to print at roughly two times the screen resolution. This format is suitable for shaded surfaces or vertex colored mesh surfaces. This is the default.
3. A value greater than 0 will be construed as an IDLgrBuffer object reference whose data will be used for printing. This format allows the programmer to control the resolution of the output of the image.

For more information on IDLgrBuffer, see the *IDL Reference Guide*.

Note

You must set the GRAPHICS_TREE property of the IDLgrWindow object for these print options to work.

The following Visual Basic example shows how to use the new BufferId property:

```
`Create an IDLgrBuffer with dimensions of 1280x1024
IDLDrawWidget1.ExecuteStr("buffer=OBJ_NEW(IDLgrBuffer, $
    dimensions=[1280,1024])")

`Get the object reference of the buffer we just created
buffer=IDLDrawWidget1.GetNamedData("buffer")

`Set the buffer ID to the object reference
IDLDrawWidget1.BufferId=buffer

`Increase the size of the buffer to 2000 pixels by 2000 pixels
IDLDrawWidget1.ExecuteStr("buffer->SetProperty(dimensions = $
    [2000,2000])")
```

Tip

Remember to destroy the IDLgrBuffer object after it is no longer needed for printing purposes.

Improved Error Reporting in the IDLDrawX2 ActiveX Control

The following have been added to aid developer's in reporting errors:

Return Value Change for ExecuteStr

The BOOL return value has been replaced by a LONG return value which is 0 if successful or the IDL error code if it fails.

Tip

This can be used in conjunction with the new LastIdlError property that contains the actual text of the error message to help you debug your program.

LastIdlError (Runtime)

A string that contains the last IDL error message. This string will not change if the ExecuteStr method is called and an error does not occur.

Tip

You can check the return value from the ExecuteStr method to determine if an error occurred. For more information, see IDLgrWindow in the *IDL Reference Guide*.

Method Enhancements to the IDLDrawX2 ActiveX Control

New Parameter for SetNamedArray

BOOL: Set to TRUE if the control should free a shared array when IDL releases its reference.

New Properties to the IDLDrawX2 ActiveX Control

Renderer

This property specifies either the software or hardware renderer for object graphics windows is to be used. It Has no effect if the GraphicsLevel property is set to 1. Valid values are:

- 0 = Platform native OpenGL
- 1 = IDL's software implementation

By default, the setting in your IDL preferences is used.

New Auto Event Properties to the IDLDrawX2 ActiveX Control

OnDbClick

An IDL procedure that will be called when a mouse button is double clicked within the draw widget. The procedure must be in the form:

```
pro button_dbclick, drawId, button, xPos, yPos
```

The following table describes each parameter of the syntax:

Parameter	Description
button	Describes which mouse button has been clicked. The valid values are: <ul style="list-style-type: none"> • 1 — Left mouse button. • 2 — Middle mouse button. • 4 — Right mouse button.
xPos	The horizontal position of the mouse when the button was clicked.
yPos	The vertical position of the mouse when the button was clicked.

Table 1-1: OnDbClick Parameters

IDL ActiveX Control Examples

All of the example IDL ActiveX control examples in the *rsi-directory*\external\activex directory (where *rsi-directory* is the installation directory for IDL) have been updated to the new IDLDrawX2 ActiveX control. These examples show the techniques you can use to create applications that use the IDL ActiveX control in several different environments.

Installation and Licensing Enhancements

IDL ActiveX Control Demonstration Application

The IDL ActiveX Control demonstration application or “Tstorm demo” is now an option for installing on Windows platforms. This application shows how the IDL ActiveX control can be used to create IDL applications. To start the application, select Start → Programs → Research Systems IDL 5.3 → TStorm.

This demonstration application shows how easy it is to create Windows applications using the IDL ActiveX control.

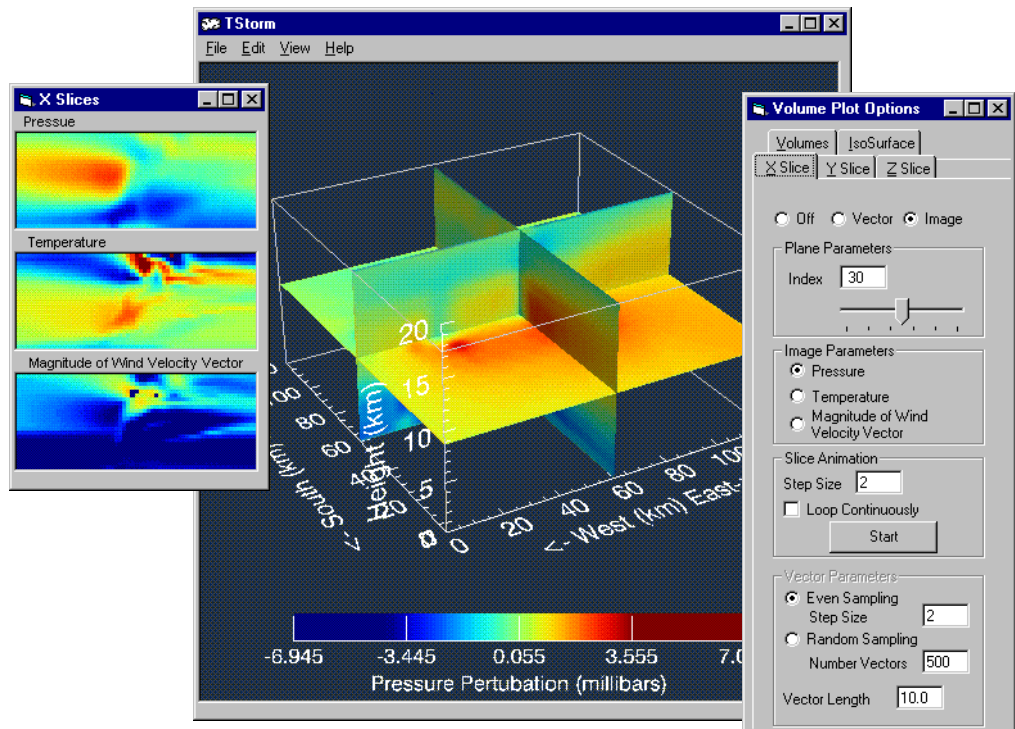


Figure 1-11: IDL ActiveX Control Demonstration Application

You can view the source for this demonstration application in the *rsi-directory/examples/tstorm* where *rsi-directory* is the installation directory for IDL.

New licensing Dialog

The IDL License Information dialog has been enhanced to ease licensing of IDL. If you are using the HASP (or Node Locked Hardware) option, you can now add any optional features listed on your RSI Registration/Licensing form in the Optional Features box.

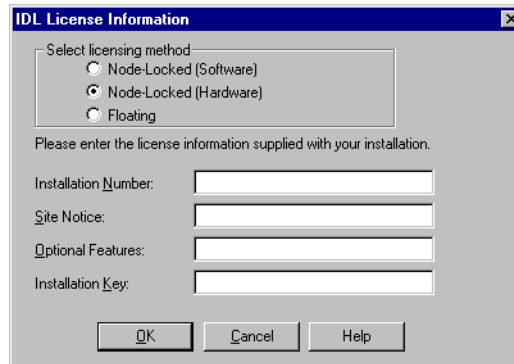


Figure 1-12: IDL License Information Dialog

For more information on how to install and license IDL, see the *Installation Guide* for your platform.

Replacing the Licensing Dialog Image in Callable IDL Applications

You can now specify the image for the Demo dialog that appears for an IDL callable application. This allows you to customize the licensing of your callable IDL application.

The Unlicensed Application dialog displays at the startup of a callable IDL application if it is not licensed.

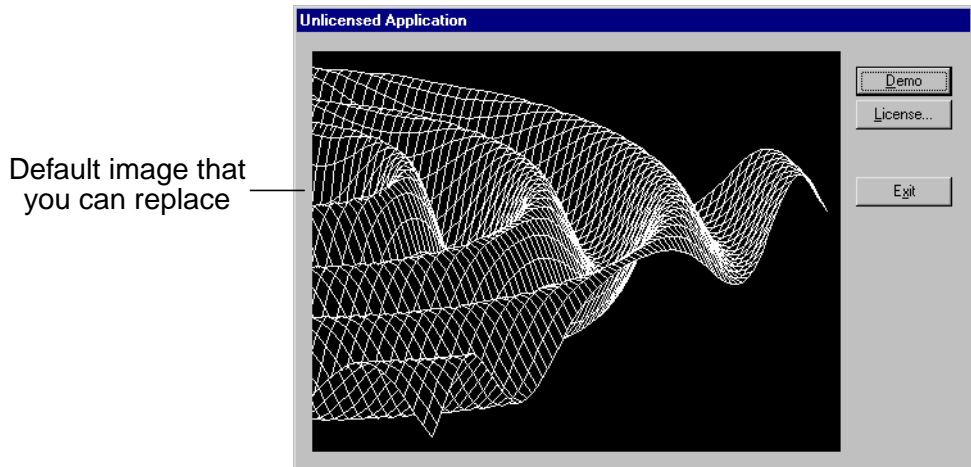


Figure 1-13: Unlicensed Application Dialog for Windows

Replacing the Image for Windows Callable Applications

To replace the image in the Unlicensed Application dialog for Windows, you use the `IDL_SetValue` routine:

```
int IDL_SetValue(int id, void* pvValue);
```

You must call the `IDL_SetValue` routine prior to the `IDL_Win32Init()` call which initializes IDL. `pvValue` may be either a string containing the path of a `.BMP` file or a bitmap resource defined in your callable application.

For example, to specify a path to a `.BMP` file, you would use something like the following:

```
// string containing path of bitmap file
strcpy(bitmapFile, "c:\\test_app\\source\\example.bmp");
IDL_SetValue(IDL_VAL_DEMODLG_BITMAP, (void*) bitmapFile);
```

If you are specifying a resource, you would use something like the following:

```
// bitmap resource
IDL_SetValue(IDL_VAL_DEMODLG_BITMAP, (void*) IDB_BITMAP1);
```

Replacing the Image for Macintosh Callable Applications

To replace the image in the Unlicensed Application dialog for Macintosh, you need to edit the IDL executable resource using a resource editor. In the following instructions, ResEdit is used to modify the resource.

To replace the image for Macintosh callable applications, complete the following steps:

1. Copy the graphic you want to add to the Unlicensed Application dialog to the clipboard.
2. Start ResEdit.
3. Open the IDL executable.
4. Open the PICT resources by double-clicking on the PICT icon.
5. Open the 139 resource by double-clicking it.
6. Paste the graphic into the window. Choose Edit → Paste.
7. Save the file. Choose File → Save.
8. Quit ResEdit. Choose File → Quit.

IDL DataMiner Enhancements

IDL DataMiner is now supported on the IRIX platform. The following table describes the supported databases:

Supported Databases	Driver Name	Platform Information
INFORMIX 5.x, 6.x, or 7.x	INFORMIX	IRIX 6.4
INFORMIX 7.x or 9.x	INFORMIX 9	IRIX 6.4
Oracle 8.0	Oracle 8	IRIX 6.4 (requires Oracle N32 Client Development Kit, Version 8.0.5.0.0 (Oracle Part Number: Z24604-02) or later)
SQL Server 4.9.2, SQL Server System 10, System 11, and Adaptive Server 11.5 and 11.9	Sybase	IRIX 6.4

Table 1-2: Supported IRIX ODBC Drivers for DataMiner

For more information on how to install, setup, and use IDL DataMiner on IRIX, see the *IDL DataMiner* manual.

Documentation Enhancements

Numerous improvements have been made to the documentation for IDL 5.3, including the printed manuals, online help, and PDF versions of each manual. This section outlines these changes, and discusses the organization of the document set and online help.

Here's a summary of what has been improved and added in the IDL 5.3 documentation.

Reorganization of Core IDL Manuals

The content of the IDL 5.3 core documentation set has been reorganized to make it easier to find the information you need. Existing IDL documentation has been enhanced visually and organizationally to help you access the full power of IDL.

IDL Reference Guide

The *IDL Reference Guide* is now a comprehensive reference for IDL that contains the following:

- An alphabetical list of IDL routines that now includes executive commands, IDL objects, and IDL statements, in addition to the already-existing functions and procedures. Any IDL language element (with the exception of Scientific Data Formats routines) that can be entered at the command prompt or in an IDL program can now be found in the *IDL Reference Guide*. Descriptions of Scientific Data Formats routines (CDF_*, EOS_*, HDF_*, and NCDF_* routines) can be found in the *Scientific Data Formats* manual.
- The Syntax (formerly called “Calling Sequence”) for each function, procedure, and object now includes all the keywords available for a routine.

Using IDL

The *Using IDL* manual now has placed a new emphasis on how to perform such powerful IDL functionality as signal processing, image processing and mapping among others.

Building IDL Applications

Building IDL Applications now has placed a new emphasis on creating applications in IDL. It covers such topics as how to create applications, components of IDL applications, and programming tools that help you build IDL applications.

Object Graphics Documentation

The *Object Graphics* manual has been eliminated, and the material has been moved into other manuals as follows:

- Information on using IDL Object Graphics has been moved to the *Using IDL* manual.
- The Object Graphics Class Library has been moved to the *IDL Reference Guide*.

The New Getting Started with IDL Manual

The new *Getting Started with IDL* manual replaces the former *IDL Basics* manual as your introduction into the world of IDL. As always, this book assumes beginning-level exposure to IDL. The existing material has been revised to be more user-friendly, following a step-by-step example-based format, and relevant new material has been added.

Improved IDL HandiGuide

The *IDL HandiGuide* now alphabetically lists all IDL functions, procedures, statements, objects, and executive commands, including Scientific Data Format routines. A description and the syntax (including keywords) is listed for each routine or object.

The New IDL Master Index

A master index to the entire IDL documentation set now exists as a PDF file and can be accessed through the IDL Online Guide. All entries are hypertext linked to the actual information you are looking for. The combined index in the back of each core manual has been removed. The index in the back of each manual now applies only to that manual.

Improved Help System

The IDL Help system has been reorganized to help you find the information you need.

The Contents tab is no longer organized according to the individual books in the IDL document set. Instead, all information in the entire IDL document set is presented by topic. This allows you to find the information you need without having to know which printed book the information resides in.

The IDL online help is now contained in a single .hlp file. This allows you to search and find the information you're looking for in one source.

The IDL online help also contains a convenient navigational section in its commands reference section to allow you to easily navigate to the routine or procedure you're looking for.

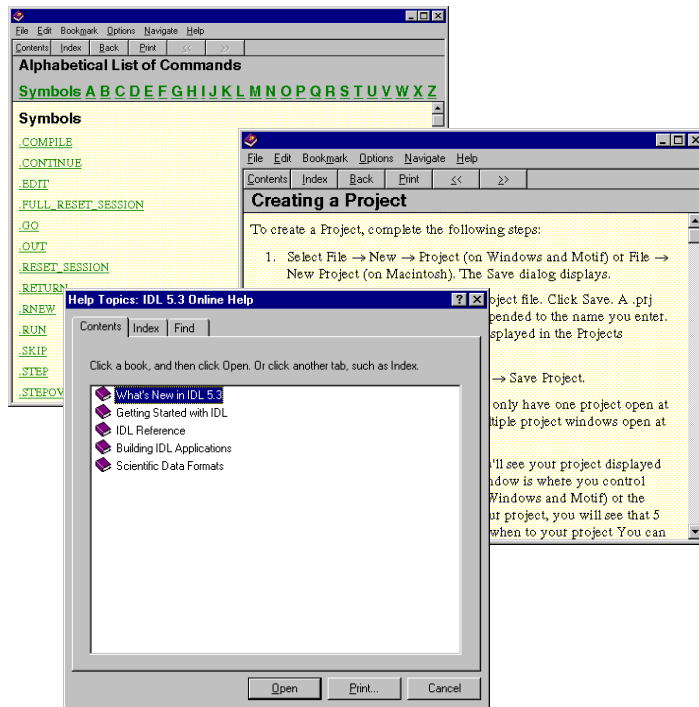


Figure 1-14: The IDL Help Navigator

Improved PDF System

All volumes of the IDL documentation set are now available in Adobe Acrobat Portable Document Format (PDF). These PDF files are automatically installed on your machine with IDL. You will need a copy of Adobe's Acrobat Reader with Search software (version 3.0 or later). A copy of Adobe Acrobat Reader is included on your product CD-ROM. For more information on Adobe Acrobat Reader, visit their World Wide Web site at www.adobe.com

To access the IDL online manuals after you have installed IDL:

- On Windows, select Start → Programs → Research Systems IDL 5.3 → IDL Online Manuals.
- On Macintosh, a shortcut can be found in the *rsi-directory:RSI:IDL 5.3* folder named *IDL Online Manuals*.
- On UNIX, execute the following at the UNIX prompt:

```
idlman
```

The IDL online manuals can also be found in the `info` directory of your product CD-ROM.

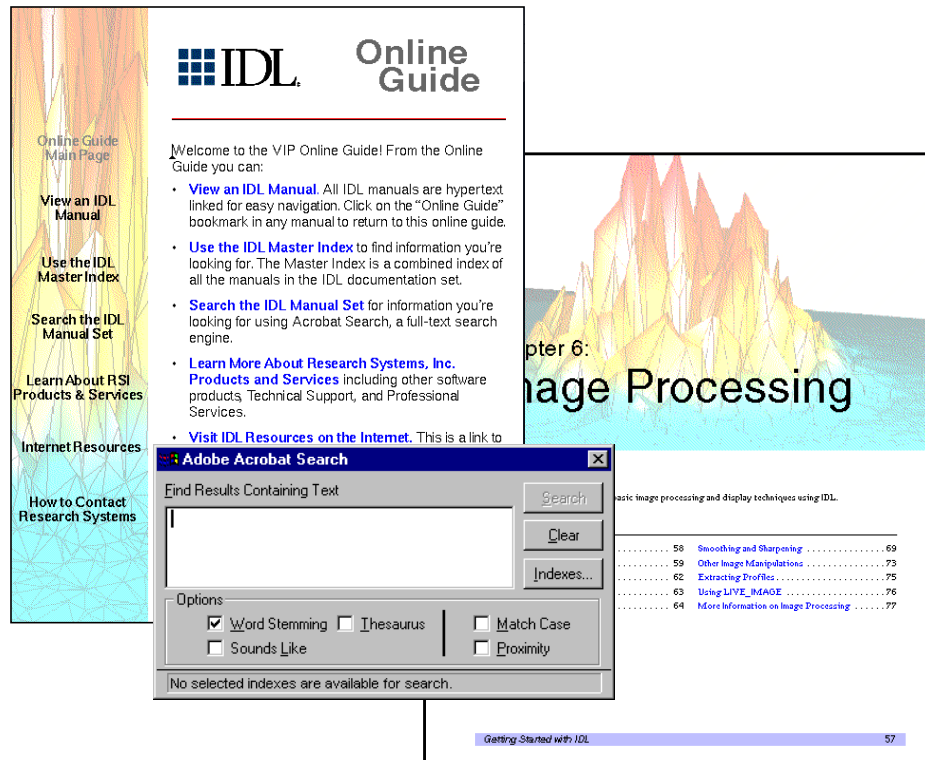


Figure 1-15: The Online Guide and Search Dialog

Navigation of the IDL Online Manuals

The online IDL manuals are fully hypertext linked for easy navigation. The Online Guide (onlguide.pdf) file is your guide to the IDL documentation set. It has links for all manuals in the documentation set as well as links on how to get more information from Research Systems.

Searching within the Online Manual Set

The IDL online manuals are set up to search for any information you might need within the IDL manual set. To search the IDL manual set, you can click on the binocular/page button in the Acrobat Reader tool bar after you have opened any IDL manual in the set including the Online Guide.

New Object Features

This section describes the new object classes, methods, and keywords in IDL 5.3.

New Object Classes

The following table describes the new object classes in IDL 5.3:

Object Class	Description
IDLanROI	The IDLanROI object class provides an analytical representation of a region of interest.
IDLanROIGroup	Analytical representation of a group of regions of interest.
IDLffLanguageCat	Provides an interface to IDL language catalog files.
IDLgrROI	An Object Graphics representation of a region of interest.
IDLgrROIGroup	Object Graphics representation of a group of regions of interest.

Table 1-3: New Object Classes in IDL 5.3.

New Object Methods

New and existing IDL Object Graphics classes have been updated to include the following new methods:

New Methods	Description
IDLanROI::AppendData	Appends vertices to the current region.
IDLanROI::Cleanup	Performs all cleanup for a region of interest object.
IDLanROI::ComputeGeometry	Computes the requested geometrical values (area, perimeter, and/or centroid) for the region.

Table 1-4: New Object Methods in IDL 5.3

New Methods	Description
IDLAnROI::ComputeMask	Prepares a two-dimensional mask for this region.
IDLAnROI::ContainsPoints	Determines whether the given data coordinates are contained within this region.
IDLAnROI::GetProperty	Retrieves the value of a property or group of properties for the region.
IDLAnROI::Init	Initializes a region of interest object.
IDLAnROI::RemoveData	Removes vertices from the region.
IDLAnROI::ReplaceData	Replaces vertices in the region with alternate values.
IDLAnROI::Rotate	Modifies the vertices for the region by applying a rotation.
IDLAnROI::Scale	Modifies the vertices for the region by applying a scale.
IDLAnROI::SetProperty	Sets the value of a property or group of properties for the region.
IDLAnROI::Translate	Modifies the vertices for the region by applying a translation.
IDLAnROIGroup::Add	Adds a region to the region group. Only objects of the IDLAnROI class may be added to the group.
IDLAnROIGroup::Cleanup	Performs all cleanup for a region of interest group object.
IDLAnROIGroup::ContainsPoints	Determines whether the given data coordinates are contained within the closed polygon regions within this group. A point is considered to be contained if it lies within the boundary of a region.
IDLAnROIGroup::ComputeMask	Prepares a two-dimensional mask for this group of regions.

Table 1-4: New Object Methods in IDL 5.3

New Methods	Description
IDLanROIGroup::ComputeMesh	Triangulates a surface mesh (optionally capped) from the stack of regions contained within this group.
IDLanROIGroup::GetProperty	Retrieves the value of a property or group of properties for the group of regions.
IDLanROIGroup::Init	Initializes a region of interest group object.
IDLanROIGroup::Rotate	Modifies the vertices for all regions within the group by applying a rotation.
IDLanROIGroup::Scale	Modifies the vertices for the region by applying a scale.
IDLanROIGroup::Translate	Modifies the vertices all regions within the group by applying a translation.
IDLffLanguageCat::IsValid	Determines whether the IDLffLanguageCat object has a valid catalog.
IDLffLanguageCat::Query	Returns the language string associated with the given key.
IDLffLanguageCat::SetCatalog	Sets the appropriate catalog file.
IDLgrBuffer::GetDeviceInfo	Returns information on OpenGL hardware that may be installed on the computer.
IDLgrClipboard::GetDeviceInfo	Returns information on OpenGL hardware that may be installed on the computer.
IDLgrROI::Cleanup	Performs all cleanup for an Object Graphics region of interest object.
IDLgrROI::GetProperty	Retrieves the value of a property or group of properties for the Object Graphics region.
IDLgrROI::Init	Initializes an Object Graphics region of interest object.

Table 1-4: New Object Methods in IDL 5.3

New Methods	Description
IDLgrROI::PickVertex	Picks a vertex of the region that, when projected onto the given destination device, is nearest to the given 2D device coordinate.
IDLgrROI::SetProperty	Sets the value of a property or group of properties for the Object Graphics region.
IDLgrROIGroup::Add	Adds a region to the region group. Only objects of the IDLgrROI class may be added to the group. The regions in the group must all be of the same type (all points, all paths, or all polygons).
IDLgrROIGroup::Cleanup	Performs all cleanup for an Object Graphics region of interest group object.
IDLgrROIGroup::Init	Initializes an Object Graphics region of interest group object.
IDLgrROIGroup::PickRegion	Picks a region within the group that, when projected onto the given destination device, is nearest to the given 2D device coordinate.
IDLgrVRML::GetDeviceInfo	Returns information on OpenGL hardware that may be installed on the computer.
IDLgrWindow::GetDeviceInfo	Returns information on OpenGL hardware that may be installed on the computer.

Table 1-4: New Object Methods in IDL 5.3

New and Updated Keywords to IDL Object Methods

The following table describes the new and updated keywords to IDL objects.

Object Method	Keyword	Description
IDLffDICOM::Read	ENDIAN	Set this keyword to configure the endian format when reading a DICOM file. <ul style="list-style-type: none"> • 1 = Implicit VR Little Endian • 2 = Explicit VR Little Endian • 3 = Implicit VR Big Endian • 4 = Explicit VR Big Endian

Table 1-5: New and Updated Keywords to IDL Object Methods in IDL 5.3

Object Method	Keyword	Description
IDLgrAxis::Init	TEXTALIGNMENTS (<i>Get, Set</i>)	<p>Set this keyword to a two-element floating point vector, [horizontal, vertical], specifying the horizontal and vertical alignments for the tick text. Each alignment value should be a value between 0.0 and 1.0. For horizontal alignment, 0.0 left-justifies the text; 1.0 right-justifies the text. For vertical alignment, 0.0 bottom-justifies the text, 1.0 top-justifies the text. The defaults are as follows:</p> <ul style="list-style-type: none"> • X-Axis: [0.5, 1.0] (centered horizontally, top-justified vertically) • Y-Axis: [1.0, 0.5] (right-justified horizontally, centered vertically) • Z-Axis: [1.0, 0.5] (right-justified horizontally, centered vertically)
	TEXTBASELINE (<i>Get, Set</i>)	<p>Set this keyword to a two- or three-element vector describing the direction in which the baseline of the tick text is to be oriented. Use this keyword in conjunction with the TEXTUPDIR keyword to specify the plane on which the tick text lies. The default is [1,0,0].</p>

Table 1-5: New and Updated Keywords to IDL Object Methods in IDL 5.3

Object Method	Keyword	Description
IDLgrAxis::Init (continued)	TEXTUPDIR (<i>Get, Set</i>)	Set this keyword to a two- or three-element vector describing the direction in which the up-vector of the tick text is to be oriented. Use this keyword in conjunction with the TEXTBASELINE keyword to specify the plane on which the tick text lies. TEXTUPDIR should be orthogonal to TEXTBASELINE. The default is as follows: <ul style="list-style-type: none"> • X-Axis: [0, 1, 0] • Y-Axis: [0, 1, 0] • Z-Axis: [0, 0, 1]
IDLgrClipboard::Draw	FILENAME	Set this keyword to a string representing the name of a file to which the output should be written. By default, this keyword is a null string indicating that the output is written to the clipboard.
	POSTSCRIPT	Set this keyword to a nonzero value to indicate that the generated output should be in PostScript format. By default, the generated output is in Windows Enhanced Metafile Format on Windows platforms, PICT format on Macintosh platforms, and PostScript on UNIX/VMS platforms.

Table 1-5: New and Updated Keywords to IDL Object Methods in IDL 5.3

Object Method	Keyword	Description
IDLgrClipboard::Draw (continued)	VECTOR	Set this keyword to indicate the type of graphics primitives generated. Valid values are: <ul style="list-style-type: none"> • 0 — Bitmap (the default). The Draw method renders the scene to a buffer and then copies the buffer to the printer in bitmap format. The bitmap retains the quality of the original image. • 1 — Vector. The Draw method renders the scene using simple vector operations that result in a representation of the scene that is scalable to the printer. This representation does not retain all the attributes of the original image.
IDLgrImage::Init	SUB_RECT (<i>Get, Set</i>)	Set this keyword to a four-element vector, [<i>x, y, xdim, ydim</i>], specifying the position of the lower left-hand corner and the dimensions of the sub-rectangle to display.

Table 1-5: New and Updated Keywords to IDL Object Methods in IDL 5.3

Object Method	Keyword	Description
IDLgrPrinter::Draw	VECTOR	<p>Set this keyword to indicate the type of graphics primitives generated. Valid values are:</p> <ul style="list-style-type: none"> • 0 — Bitmap (the default). The Draw method renders the scene to a buffer and then copies the buffer to the printer in bitmap format. The bitmap retains the quality of the original image. • 1 — Vector. The Draw method renders the scene using simple vector operations that result in a representation of the scene that is scalable to the printer. The vector representation does not retain all the attributes of the original image. The vector representation is sent to the printer.

Table 1-5: New and Updated Keywords to IDL Object Methods in IDL 5.3

New Routines

The following is a list of new functions, procedures, statements, and executive commands added to IDL. To view documentation for new routines, see [Chapter 5](#), “New IDL Routines”.

Routine	Description
.RESET_SESSION	<p>This executive command resets much of the state of an IDL session without requiring the user to exit and restart the IDL session. Note that executive commands can only be used at the IDL command prompt, not in IDL programs.</p>
.FULL_RESET_SESSION	<p>This executive command does everything .RESET_SESSION does, plus the following:</p> <ul style="list-style-type: none"> • Removes all system routines installed via LINKIMAGE or a DLM. • Removes all structure definitions installed via a DLM. • Removes all message blocks added by DLMs. • Unloads all sharable libraries loaded into IDL via CALL_EXTERNAL, LINKIMAGE, or a DLM. • Re-initializes all DLMs to their unloaded initial state. <p>Note that executive commands can only be used at the IDL command prompt, not in IDL programs.</p>
ADAPT_HIST_EQUAL	<p>Performs adaptive histogram equalization, a form of automatic image contrast enhancement.</p>

Table 1-6: New Routines in IDL 5.3

Routine	Description
BINARY_TEMPLATE	Allows the user to interactively generate a template structure for use with <code>READ_BINARY</code> .
CDF_COMPRESSION	Sets or returns the compression mode for a CDF file and/or variables.
COMPILE_OPT	Allows the author to provide the IDL compiler with information that changes some of the default rules for how to compile the function or procedure within which the <code>COMPILE_OPT</code> statement appears.
CW_FILESEL	A compound widget for file selection.
CW_LIGHT_EDITOR	Creates a compound widget to edit properties of existing <code>IDLgrLight</code> objects in a view.
CW_LIGHT_EDITOR_GET	Gets the <code>CW_LIGHT_EDITOR</code> properties.
CW_LIGHT_EDITOR_SET	Sets the <code>CW_LIGHT_EDITOR</code> properties.
CW_PALETTE_EDITOR	Creates a compound widget to display and edit color palettes.
CW_PALETTE_EDITOR_GET	Gets the <code>CW_PALETTE_EDITOR</code> properties.
CW_PALETTE_EDITOR_SET	Sets the <code>CW_PALETTE_EDITOR</code> properties.
DIALOG_READ_IMAGE	A graphical user interface used for reading image files.
DIALOG_WRITE_IMAGE	A graphical user interface used for writing image files.
DLM_LOAD	Normally, IDL system routines that reside in Dynamically Loadable Modules (DLMs) are automatically loaded on demand when a routine from a DLM is called. The <code>DLM_LOAD</code> procedure can be used to explicitly cause a DLM to be loaded.
DRAW_ROI	Draws a region to the current Direct Graphics device.

Table 1-6: New Routines in IDL 5.3

Routine	Description
ENABLE_SYSRTN	Enables/disables IDL system routines.
EOS_EXISTS	Returns success (1) if the HDF-EOS extensions are supported on the current platform, and fail (0) if not.
EOS_GD_QUERY	Returns information about a specified grid.
EOS_PT_QUERY	Returns information about a specified point.
EOS_QUERY	Returns information about the makeup of an HDF-EOS file.
EOS_SW_QUERY	Returns information about a specified swath.
GET_DRIVE_LIST	Returns a string array of the names of valid drives / volumes for the file system. (Windows / Macintosh only)
GRID_TPS	Uses thin plate splines to interpolate a set of values over a regular grid.
HDF_SD_SETCOMPRESS	Compresses an existing HDF SD data set or sets the compression method of a newly created HDF SD data set.
IMAGE_STATISTICS	Computes sample statistics for a given array of values.
ISOCONTOUR	Allows for contouring on arbitrary meshes and returns line or orientated tessellated polygon output.
ISOSURFACE	Returns topologically consistent triangles by using orientated tetrahedral decomposition internally and allows the algorithm to isosurface any arbitrary tetrahedral mesh.
MESH_CLIP	Clips a polygon mesh to an arbitrary plane in space and returns a polygon mesh of the remaining portion.

Table 1-6: New Routines in IDL 5.3

Routine	Description
MESH_DECIMATE	Accepts an additional array of auxiliary data values which can be used together with a weighting function to allow external data to be considered when a particular triangle is removed.
MESH_ISSOLID	Computes various mesh properties and enables IDL to determine if a mesh is a solid. If the mesh can be considered a solid, routines can compute the volume of the mesh.
MESH_MERGE	Merges two polygonal meshes.
MESH_NUMTRIANGLES	Computes various mesh properties and enables IDL to determine the number of triangles in a polygonal mesh.
MESH_SMOOTH	Performs spatial smoothing on a polygonal mesh.
MESH_SURFACEAREA	Computes various mesh properties and enables IDL to determine the mesh surface area, including integration of other properties interpolated on the surface of the mesh.
MESH_VALIDATE	Checks for NaN values in vertices and removes unused vertices.
MESH_VOLUME	Computes various mesh properties and enables IDL to determine the volume that the mesh encloses.
MORPH_CLOSE	Applies the closing operator to a binary or grayscale image.

Table 1-6: New Routines in IDL 5.3

Routine	Description
MORPH_DISTANCE	Estimates N -dimensional distance maps, which contain for each foreground pixel the distance to the nearest background pixel, using a given norm. Available norms include: Euclidean, which is exact and is also known as the Euclidean Distance Map (EDM), and two more efficient approximations, chessboard and city block.
MORPH_GRADIENT	Applies the morphological gradient operator to a grayscale image.
MORPH_HITORMISS	Applies the hit-or-miss operator to a binary image.
MORPH_OPEN	Applies the opening operator to a binary or grayscale image.
MORPH_THIN	Implements a thinning operator on binary images.
MORPH_TOPHAT	Applies the top-hat operator to a grayscale image.
MSG_CAT_CLOSE	Closes an IDL language catalog file from the stored cache.
MSG_CAT_COMPILE	Creates an IDL language catalog file.
MSG_CAT_OPEN	Returns a specified object from an IDL language catalog file.
PARTICLE_TRACE	Traces the path of a mass-less particle through a vector field and allows the user to specify a set of starting points and a vector field.
QUERY_IMAGE	Reads the header of a file and determines if it is recognized as an image file.
QUERY_WAV	Reads the header of a file and determines if it is recognized as a .WAV file.

Table 1-6: New Routines in IDL 5.3

Routine	Description
READ_BINARY	Reads the contents of a binary file using a passed template or basic command line keywords.
READ_IMAGE	Reads the image contents of a file.
READ_WAV	Reads the audio stream from the named .WAV file.
STRCMP	Compares two strings. Can perform case-insensitive comparison of first N characters more easily than using the EQ operator.
STREAMLINE	Computes a line that traces the path of a particle through a constant vector field.
STREGEX	Performs regular expression string matching.
STRJOIN	Collapses a string scalar or array into merged strings. The separator string used between the joined strings can be specified.
STRMATCH	Compares its search string, which can contain wildcard characters, against the input string expression.
STRSPLIT	Splits its input string argument into separate substrings, according to the specified delimiter or regular expression.
STRUCT_HIDE	Used by authors of large vertical applications to prevent the IDL HELP procedure from displaying information about structures or objects that are not part of their public interface.
TETRA_CLIP	Clips a tetrahedral mesh to an arbitrary plane in space and return a tetrahedral mesh of the remaining portion.
TETRA_SURFACE	Extracts a polygonal mesh as the exterior surface of a tetrahedral mesh.

Table 1-6: New Routines in IDL 5.3

Routine	Description
TETRA_VOLUME	Computes properties of a tetrahedral mesh array.
VALUE_LOCATE	Finds the interval(s) within a given monotonically increasing (or monotonically decreasing) vector that brackets a given search value (or set of values).
VECTOR_FIELD	Used to place colored, orientated vectors of specified length at each vertex in an input vertex array.
WATERSHED	Applies watershed segmentation to a binary image.
WRITE_IMAGE	Writes an image and its color table vectors, if any, to a file of a specified type.
WRITE_WAV	Writes the audio stream from the named .WAV file.
XOBJVIEW	Allows you to quickly and easily view and manipulate IDL Object Graphics on screen. This procedure displays a widget containing buttons that allow you to rotate, pan, and scale the object using your mouse.

Table 1-6: New Routines in IDL 5.3

New and Updated Keywords/Arguments

The following is a list of new and updated keywords and arguments to existing IDL routines.

Routine	Keyword/ Argument	Description
BREAKPOINT	DISABLE	Disables the specified breakpoint if it exists. The breakpoint can be specified using the breakpoint index or file and line number.
	ENABLE	Enables the specified breakpoint if it exists. The breakpoint can be specified using the breakpoint index or file and line number.
CALL_EXTERNAL	UNLOAD	Normally, IDL keeps <i>Image</i> loaded in memory after the call to CALL_EXTERNAL completes. This is done for efficiency—loading a sharable object can be a slow operation. Setting the UNLOAD keyword will cause IDL to unload <i>Image</i> after the call to it is complete.
CHECK_MATH	MASK	Defines the bitmask of exceptions to check.
	NOCLEAR	If NOCLEAR is set, exceptions are not cleared and remain pending.

Table 1-7: New and Updated Keywords/Arguments in IDL 5.3

Routine	Keyword/ Argument	Description
DILATE	BACKGROUND	Set this keyword to the pixel value that is to be considered the background when dilation is being performed in constrained mode.
	CONSTRAINED	If this keyword is set and grayscale dilation has been selected, the dilation algorithm will operate in constrained mode. In this mode, a pixel is set to the value determined by normal grayscale dilation rules in the output image only if the current value destination pixel value matches the BACKGROUND pixel value. Once a pixel in the output image has been set to a value other than the BACKGROUND value, it cannot change.
	PRESERVE_TYPE	Set this keyword to return the same type as the input array. This keyword only applies if the GRAY keyword is set.
	UINT	Set this keyword to return an unsigned integer array. This keyword only applies if the GRAY keyword is set.
	ULONG	Set this keyword to return an unsigned longword integer array. This keyword only applies if the GRAY keyword is set.

Table 1-7: New and Updated Keywords/Arguments in IDL 5.3

Routine	Keyword/ Argument	Description
ERODE	PRESERVE_TYPE	Set this keyword to return the same type as the input array. This keyword only applies if the GRAY keyword is set.
	UINT	Set this keyword to return an unsigned integer array. This keyword only applies if the GRAY keyword is set.
	ULONG	Set this keyword to return an unsigned longword integer array. This keyword only applies if the GRAY keyword is set.

Table 1-7: New and Updated Keywords/Arguments in IDL 5.3

Routine	Keyword/ Argument	Description
EXTRACT_SLICE	ANISOTROPY	Set this keyword to a three-element array. This array specifies the spacing between the planes of the input volume in grid units of the (isotropic) output image.
	VERTICES	Set this keyword to a named variable in which to return a [3,Xsize,Ysize] floating point array. This is an array of the x, y, z sample locations for each pixel in the normal output.
	PlaneNormal	This new argument is a 3-element array that provides an alternate form for the plane specification. The values are interpreted as the normal of the slice plane.
	Xvec	This new argument is a 3-element array that provides an alternate form for the plane specification. The three values are interpreted as the 0 dimension directional vector. This should be a unit vector.

Table 1-7: New and Updated Keywords/Arguments in IDL 5.3

Routine	Keyword/ Argument	Description
FIX	PRINT	Set this keyword to specify that any special-case processing when converting between string and byte data, or the reverse, should be suppressed.
	TYPE	FIX normally converts its expression to the integer type. If TYPE is specified, it is the type code to set the type of the conversion. This feature allows dynamic type conversion, where the desired type is not known until runtime, to be carried out without the use of large CASE or IF...THEN logic.
FSTAT	n/a	FSTAT returns two new fields: XDR (nonzero if the file was opened with the XDR keyword) and COMPRESS (nonzero if the file was opened with the COMPRESS keyword).
HDF_BROWSER	n/a	VGroups and VData have been added to the display options.
	n/a	A new Show3 preview type has been added, which combines an image, surface plot, and contour plot.

Table 1-7: New and Updated Keywords/Arguments in IDL 5.3

Routine	Keyword/ Argument	Description
HELP	BRIEF	BRIEF produces very terse summary style output instead of the output normally displayed by the following keywords: DLM, HEAP_VARIABLES, MESSAGES, OBJECTS, ROUTINES, SOURCE_FILES, STRUCTURES, and SYSTEM_VARIABLES.
	FULL	By default, HELP filters its output in an attempt to only display information likely to be of use to the IDL end user. Specify FULL to see all available information on a given topic without any such filtering.
	FUNCTIONS	Normally, the ROUTINES or SOURCE_FILES keywords produce information on both functions and procedures. If FUNCTIONS is specified, only output on functions is produced.

Table 1-7: New and Updated Keywords/Arguments in IDL 5.3

Routine	Keyword/ Argument	Description
HELP (<i>continued</i>)	NAMES	NAMES now works with the output from the following keywords: DLM, HEAP_VARIABLES, MESSAGES, OBJECTS, ROUTINES, SOURCE_FILES, STRUCTURES, and SYSTEM_VARIABLES.
	PROCEDURES	Normally, the ROUTINES or SOURCE_FILES keywords produce information on both functions and procedures. If PROCEDURES is specified, only output on procedures is produced.
INTERPOL	LSQUADRATIC	If set, interpolate using a least squares quadratic fit to the equation $y = a + bx + cx^2$, for each 4 point neighborhood ($x[i-1]$, $x[i]$, $x[i+1]$, $x[i+2]$) surrounding the interval of the interpolate, $x[i] \leq u < x[i+1]$.
	QUADRATIC	If set, interpolate by fitting a quadratic $y = a + bx + cx^2$, to the three point neighborhood ($x[i-1]$, $x[i]$, $x[i+1]$) surrounding the interval $x[i] \leq u < x[i+1]$.
	SPLINE	If set, interpolate by fitting a cubic spline to the 4 point neighborhood ($x[i-1]$, $x[i]$, $x[i+1]$, $x[i+2]$) surrounding the interval, $x[i] \leq u < x[i+1]$.

Table 1-7: New and Updated Keywords/Arguments in IDL 5.3

Routine	Keyword/ Argument	Description
INTERPOLATE	n/a	Expanded to allow for the interpolation of quantities of higher dimension (minimally vectors) which allows one to interpolate and sample many points at once. It also allows for multiple dimensions to be considered simultaneously.
LABEL_REGION	ALL_NEIGHBORS	Set this keyword to indicate that all adjacent neighbors to a given pixel should be searched. (This is sometimes called 8-neighbor searching when the image is 2-dimensional). The default is to search only the neighbors that are exactly one unit in distance from the current pixel (sometimes called 4-neighbor searching when the image is 2-dimensional).
	Data	This argument, which used to be called Image, can now be an n-dimensional array.
	EIGHT	This keyword is now obsolete. It has been replaced by the ALL_NEIGHBORS keyword (because this routine now handles N-dimensional data).
	ULONG	Set this keyword to specify that the output array should be an unsigned long integer.

Table 1-7: New and Updated Keywords/Arguments in IDL 5.3

Routine	Keyword/ Argument	Description
LMGR	EXPIRE_DATE	Set this keyword to a named variable that will receive a string containing the expiration date of the current IDL session if the session is a trial session. This named variable will be undefined if the IDL session has a permanent license.
	INSTALL_NUM	Set this keyword to a named variable that will receive a string containing the installation number of the current IDL session. This named variable will be undefined if the IDL session is unlicensed.
	SITE_NOTICE	Set this keyword to a named variable that will receive a string containing the site notice of the current IDL session. This named variable will be undefined if the IDL session is unlicensed.
MIN_CURVE_SURF	DOUBLE	Set this keyword to force the computation to be done in double-precision arithmetic.
OBJ_CLASS	Arg	This argument is now optional. If specified, OBJ_CLASS works as before, but if Arg is omitted, OBJ_CLASS returns an array containing the names of all known object classes.

Table 1-7: New and Updated Keywords/Arguments in IDL 5.3

Routine	Keyword/ Argument	Description
OPEN Procedures (OPENR, OPENW, OPENU)	COMPRESS	If COMPRESS is set, IDL reads and writes all data to the file in the standard GZIP format. IDL's GZIP support is based on the freely available ZLIB library by Mark Adler and Jean-loup Gailly. This means that IDL's compressed files are 100% compatible with the widely available gzip and gunzip programs.
RESOLVE_ROUTINE	EITHER	If set, indicates that the caller does not know whether the supplied routine names are functions or procedures, and will accept either.
	NO_RECOMPILE	Normally, RESOLVE_ROUTINE compiles all specified routines even if they have already been compiled. Setting NO_RECOMPILE indicates that such routines are not recompiled.

Table 1-7: New and Updated Keywords/Arguments in IDL 5.3

Routine	Keyword/ Argument	Description
ROUTINE_INFO	DISABLED	Set this keyword to get the names of currently disabled system procedures or functions (in conjunction with the FUNCTIONS keyword). Use of DISABLED implies SYSTEM, since user routines cannot be disabled.
	ENABLED	Set this keyword to get the names of currently enabled system procedures or functions (in conjunction with the FUNCTIONS keyword). Use of ENABLED implies SYSTEM, since user routines cannot be disabled.
	SOURCE	This keyword now returns the path to the SAVE file if the routine comes from an SAVE/RESTORE file.
SAVE	COMPRESS	If COMPRESS is set, IDL writes all data to the SAVE file using the ZLIB compression library to reduce its size.

Table 1-7: New and Updated Keywords/Arguments in IDL 5.3

Routine	Keyword/ Argument	Description
SPRSIN	Columns	A vector containing the column subscripts of the nonzero elements. Values must be in the range of 0 to (N-1).
	Rows	A vector, of the same length as Column, containing the row subscripts of the nonzero elements. Values must be in the range of 0 to (N-1).
	Values	A vector, of the same length as Column, containing the values of the non-zero elements.
	N	The size of the resulting sparse matrix.
STRMID	First_Character	The First_Character argument can now be an array.
	Length	The Length argument can now be an array.
	REVERSE_OFFSET	Specifies that First_Character should be counted from the end of the string backwards. This allows simple extraction of strings from the end.

Table 1-7: New and Updated Keywords/Arguments in IDL 5.3

Routine	Keyword/ Argument	Description
STRPOS	REVERSE_OFFSET	Normally, the value of the <i>Pos</i> argument is used as an offset from the beginning of the expression towards the end. Set REVERSE_OFFSET to use it as an offset from the last character of the string moving towards the beginning. This keyword makes it easy to position the starting point of the search at a fixed offset from the end of the string.
	REVERSE_SEARCH	STRPOS usually starts at <i>Pos</i> and moves toward the end of the string looking for a match. If REVERSE_SEARCH is set, the search instead moves towards the beginning of the string.
STRUCT_ASSIGN	NOZERO	Normally, any fields found in the destination structure that are not found in the source structure are zeroed. Set NOZERO to prevent this action and leave the original contents of such fields unchanged.

Table 1-7: New and Updated Keywords/Arguments in IDL 5.3

Routine	Keyword/ Argument	Description
THIN	NEIGHBOR_COUNT	Set this keyword to select an alternate form of output. In this form, output pixel values count the number of neighbors an individual skeletal pixel has (including itself). For example, a pixel that is part of a line will have the value 3 (two neighbors and itself). Terminal pixels will have the value 2, while isolated pixels have the value 1.
	PRUNE	If the PRUNE keyword is set, pixels with single neighbors are removed interactively until only pixels with 2 or more neighbors exist. This effectively removes (or “prunes”) skeleton branches, leaving only closed paths.
TOTAL	CUMULATIVE	If this keyword is set, the result is an array of the same size as the input, with each element, <i>i</i> , containing the sum of the input array elements 0 to <i>i</i> . This keyword also works with the Dimension parameter, in which case the sum is performed over the given dimension.
TRIANGULATE	n/a	Extended to allow for direct interpolation of values in an irregular grid in multiple dimensions.

Table 1-7: New and Updated Keywords/Arguments in IDL 5.3

Routine	Keyword/ Argument	Description
TRIGRID	n/a	Extended to allow for direct interpolation of values in an irregular grid in multiple dimensions.

Table 1-7: New and Updated Keywords/Arguments in IDL 5.3

New Environment Variables

The following environment variables have been added in IDL 5.3:

Environment Variable	Description
IDL_TMPDIR	<p>IDL, and code written in the IDL language, sometimes needs to create temporary files. The location where these files should be created is highly system dependent, and local user conventions are often different from “standard practice”. By default, IDL selects a reasonable location based on operating system and vendor conventions. Set the IDL_TMPDIR environment variable to override this choice and explicitly specify the location for temporary files.</p> <p>The GETENV system function handles IDL_TMPDIR as a special case, and can be used by code written in IDL to obtain the temporary file location.</p>

Table 1-8: New Environment Variables in IDL 5.3

Routines Obsoleted in IDL 5.3

The following routines were present in IDL Version 5.2 but became obsolete in IDL Version 5.3. These routines have been replaced with new routines or new keywords to existing routines that offer enhanced functionality. These obsoleted routines should not be used in new IDL code.

Routine	Replaced by	.pro File?
HDF_DFSD_* Routines	HDF_SD_* Routines	
RSTRPOS	STRPOS, /REVERSE_SEARCH	rstrpos.pro
STR_SEP	STRSPLIT for single character delimiters STRSPLIT, /REGEX for longer delimiters	str_sep.pro

Table 1-9: Routines Obsoleted in IDL 5.3

Platforms Supported in this Release

IDL supports the following platforms and operating systems:

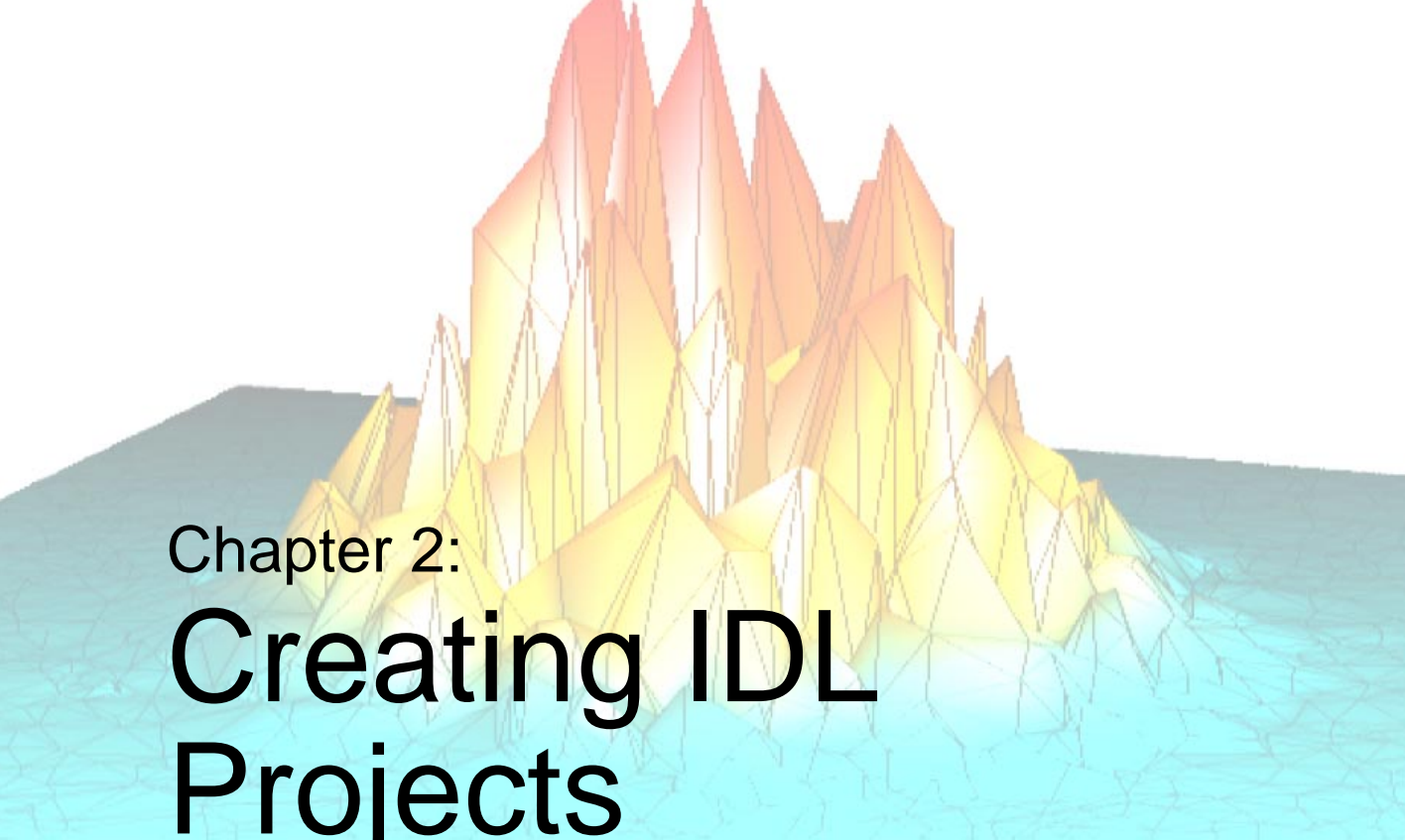
Platform	Vendor	Hardware	Operating System	Supported Versions
VMS	Compaq	Alpha	VMS	7.1
UNIX	Compaq	Alpha	Tru64 UNIX	4.0
	HP	PA-RISC	HP-UX	10.20, 11
	IBM	RS/6000	AIX	4.3
	Intel	Intel x86	Linux	2.2 †
	SGI	Mips	Irix	6.4, 6.5
	SUN	SPARC	Solaris 2	2.6, 2.7
	SUN	Ultra	Solaris 2	2.6
	SUN	Intel x86	Solaris 2	2.6
Windows	Microsoft	Intel x86	Windows	95, 98
	Microsoft	Intel x86	Windows NT	4.0
	Microsoft	Alpha	Windows NT	4.0 ††
Macintosh	Apple	PowerPC †††	MacOS	8.1

Table 1-10: Platforms Supported in IDL 5.3

†Red Hat 6.0

††IDL 5.3 is the last release that will support the Windows NT for Alpha platform.

††† Includes iMac and G3



Chapter 2: Creating IDL Projects

This chapter describes the following topics.

Overview	84	Selecting the Build Order	99
Where to Store Source Files for a Project ..	86	Running an Application from a Project ...	104
Creating a Project	87	Compiling an Application from a Project .	101
Opening, Closing, and Saving Projects	89	Building a Project	102
Adding, Moving, and Removing Files	90	Exporting a Project	105
Working with Files in a Project	93	About IDL Developer's Kit Licenses	107
Setting the Options for a Project	96		

Overview

IDL Projects allow you to easily develop applications in IDL. You can manage, compile, run, and create distributions of all the files you will need to develop your IDL application. All of your application files can be organized so that they are easier to access and easier to export to other developers, colleagues, or users. IDL Projects are a great benefit to development teams working on a large project as well as individual developers managing multiple projects.

Access to all Files in Your Application

IDL Projects have an easy to use interface for grouping:

- IDL source code files (.pro)
- GUI files (.prc) created with IDL GUIBuilder
- Data files (ASCII text or binary)
- Image files (.tif, .gif, .bmp, etc.)
- Other files (help files, .sav files, etc.)

After you add all of your files to your project, you can simply double click on .pro files to open them in the IDL editor or .prc files to open them in the IDL GUIBuilder.

Working with Files in Your Project

IDL projects makes it easy to add, remove, move, edit, compile, and test files in your project.

All of your workspace information is saved as well. If you save and exit your project with open files, when you open your project, those same files will be opened automatically for you.

IDL projects also store and retain breakpoint information. There is no need to reset breakpoints every time you open the project.

Compiling and Running Your Application

Compiling and running applications is fast and easy. You can compile all of your source files or just the files that you have modified and then run your application through the Projects menu. You can customize how your application is compiled and run by specifying options for your project.

Creating IDL Runtime Distributions

Once you have completed your application, you can quickly and easily create an IDL Runtime distribution. If you have purchased the IDL Developer's Kit, your application is automatically licensed for distribution.

Exporting Your Applications

You can easily move your application to another platform or distribute your source code to colleagues by exporting your project. All your source code, GUI files, data files, and image files are copied to a directory you specify. You can also specify to export the Runtime version of IDL with you application.

Example of a Project

A working example project has been included with IDL in the *examples* directory and is named `demo_proj.prj`.

Where to Store Source Files for a Project

The directory structure you use for your application files is important for exporting projects. Even though you can add any file from any path to your project, keep the following in mind:

- Create a directory structure with all of your files in your project. For example, you might create a directory structure similar to the following:

```
C:\myproject
  myproj.prj
  \source
  \gui
  \data
  \bitmaps
  \other
```

where all of your source files (`.pro`) are in the `source` directory, IDL GUIBuilder files (`.prc`) are in the `gui` directory, data files are in the `data` directory, image files are in the `bitmaps` directory, and any other miscellaneous files are in the `other` directory.

- Keep the project file (`.prj`) at the root level of all the other files in your project. As shown in the previous example, the project file `myproj.prj` is in the root level directory `myproject`.

When a project is exported, the files will be placed according to where they are in relation to the `.prj` file keeping the directory structure intact whenever possible. If for example, one of your source files exists in `D:\otherproj`, when you export your project it will be placed in the same directory as your project file. In this case, `C:\myproject`.

Note

The `.prj` file is not exported.

For more information on exporting a project, see [“Exporting a Project”](#) on page 105.

Creating a Project

To create a Project, complete the following steps:

1. Select **File** → **New** → **Project** (on Windows and Motif) or **File** → **New Project** (on Macintosh). The Save dialog displays.
2. Select the path and name of the project file. Click **Save**. A `.prj` extension will automatically be appended to the name you enter. You will see that your project is displayed in the **Projects Window**.

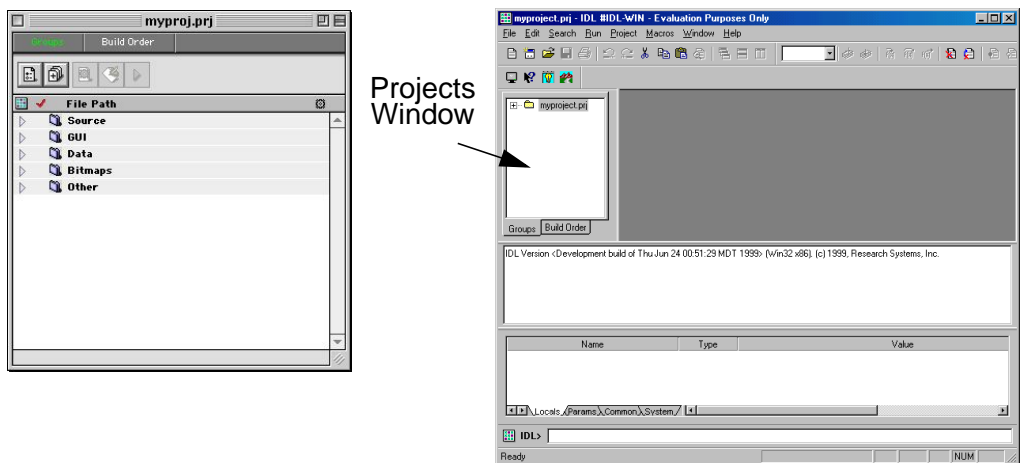


Figure 2-1: Projects Window for Macintosh (left) and Windows (right)

3. Save your new project. Select **File** → **Save Project**.

Note

For Windows and Motif, you can only have one project open at a time. On Macintosh, you can have multiple project windows open at the same time.

After you have created your project, you'll see your project displayed in the Projects Window. The Projects window is where you control your project. If you click the plus sign (Windows and Motif) or the expand arrow (Macintosh) to expand your project,

you will see that 5 groups have been automatically created when you created your project. You can then click the minus sign to collapse the listing.

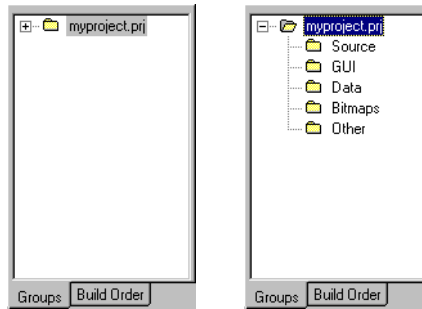


Figure 2-2: Project Window Collapsed (Left) and Expanded (Right)

The following table describes the purpose for each group:

Group	Description
Source	Stores IDL source code files (.pro).
GUI	Stores GUI files (.prc) created using the IDL GUIBuilder.
Data	Stores any data files (ASCII or binary).
Bitmaps	Stores bitmap files (.gif, .tif, .bmp, etc.).
Other	Stores any other files that do not apply to the other groups.

Table 2-1: Project Group Descriptions

Opening, Closing, and Saving Projects

After you have created a project, you can open, save, and close a project.

Opening Projects

To open a project, complete the following steps:

1. For Windows and Motif, select **File** → **Open Project**. For Macintosh, select **File** → **Open**.
2. Select the path and name of your project file.

Tip

IDL keeps track of the most recently opened projects. You can use the **File** → **Recent Projects** menu (on Windows and Motif) and **File** → **Open Recent** (on Macintosh) to select projects to open.

Saving Projects

To save a project, select **File** → **Save Project**.

Tip

IDL Projects stores information about the project's workspace. This includes information about which files you have open and breakpoints you have set. If you have files that you are commonly working in all the time, leave those files open when saving and closing your project. These files will be automatically opened in the IDL Editor or GUIBuilder windows when you reopen your project.

Closing Projects

To close a project, select **File** → **Close Project**.

Adding, Moving, and Removing Files

After you have created a project, you can add, move, and remove files in your application.

Adding Files

To add files to your project, complete the following steps:

1. Open your project. Select **File** → **Open Project**. Select the path and name of your project file.
2. Click **Project** → **Add/Remove Files...** (on Windows and Motif) or **Project** → **Add Files...** (on Macintosh). The **Add/Remove Files** dialog displays.

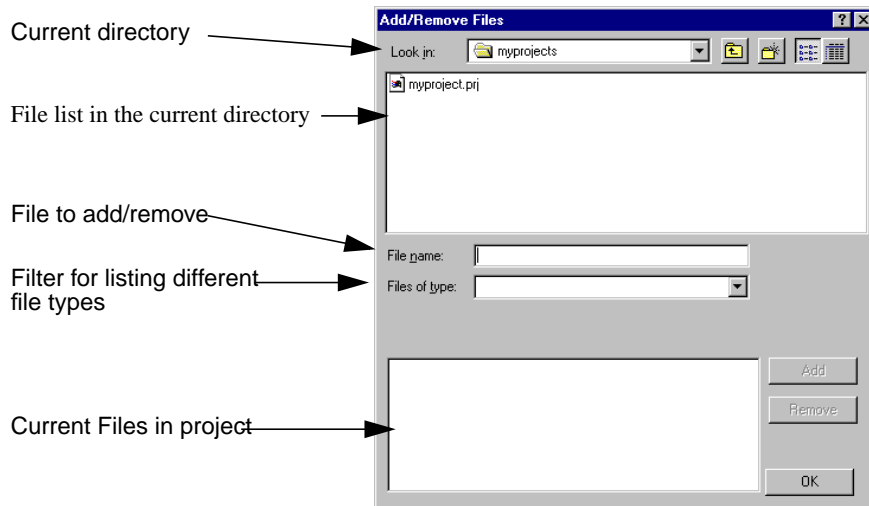


Figure 2-3: Add/Remove Dialog

3. Select the path and name of the file you want to add to your project. Click the **Add** button. You will see the file added to the list of current files in your project.

Tip

You can also add files to your project by dragging and dropping the files from any file manager. If you already have the file open that you want to add to your project,

you can right click in the editor window and select **Add to Current Project** from the context menu. On the Macintosh, you can also add an open file to your project by selecting **Project** → **Add Window**. On some Motif platforms, dragging and dropping is not supported. In this case, use the **Add/Remove...** dialog.

4. Continue to add the files you want to include in your project. Then click **OK**.
5. You can expand the listings in the Project window to see the files you have added.
6. Save your project file by selecting **File** → **Save Project**.

Moving Files

When you add a file to your project, it will be added to the appropriate group. If you want the file to exist in a different group, you can move it to that group. To move a file, complete the following steps:

1. Open your project. Select **File** → **Open Project** (on Windows and Motif) or **File** → **Open** (on Macintosh). Select the path and name of your project file.
2. Click on the plus sign (on Windows and Motif) or the expand arrow (on Macintosh) to expand the listing of the project files until you see the file you want to move.
3. To move the file, select the file and then drag it to a different group or right click over the file you want to move and select **Move To...** from the context menu and then select the different group.

Note

On some Motif platforms, dragging and dropping is not supported. In this case, use the **Move To...** menu item on the context menu.

4. Save your project file by selecting **File** → **Save Project**.

Note

When moving a file in your project, it does not change the actual path of the file, it only moves it with respect to which group the file appears within your project.

Removing Files

When you no longer want a file to be in your project, you can remove it. When you remove a file from your project, it does not delete the file on your disk, it only deletes the reference to the file from your project.

On Windows and Motif, to remove files from your project, complete the following steps:

1. Open your project. Select **File** → **Open Project** and select the path and name of your project file.
2. Click **Project** → **Add/Remove Files...** (on Windows and Motif) or **Project** → **Add Files...** (on Macintosh). The **Add/Remove Files** dialog displays.
3. Click on the file you want to remove from your project in the current files listing. Click **Remove**.

Tip

On Windows and Motif, you can use the context menu to remove a file. Right click over the file and then select **Remove**. On Windows, you can also use the Delete key to remove files. Select the file by left clicking over the file and then press the Delete key.

4. Save your project file by selecting **File** → **Save Project**.

On Macintosh, to remove files from your project, complete the following steps:

1. Open your project. Select **File** → **Open** and select the path and name of your project file.
2. Select the file you want to remove.
3. Select **Project** → **Remove Selection**.

Tip

On Macintosh, you can use the Cmd-Delete key sequence to remove files. Select the file by clicking over the file and then press Cmd-Delete.

4. Save your project file by selecting **File** → **Save Project**.

Working with Files in a Project

Once you have added all of the files in your application to a project, you can access those files through the project.

Editing a Source File

All source files that can be opened in IDL (.pro and .prc files) can be opened directly through the project. To open a file for editing, complete the following steps:

1. Open your project. Select **File** → **Open Project** (on Windows and Motif) or **File** → **Open** (on Macintosh). Select the path and name of your project file.
2. Access the context menu by right clicking over the file you want to open. Select **Edit** from the context menu. Source files (.pro) are opened in the IDL editor and GUIBuilder files (.prc) are opened in the IDL GUIBuilder

Tip

Double clicking on the file will also open .pro and .prc files. Also, on Windows you can drag the file from the Projects window to the IDL Editor window to open the file.

Compiling a File

All source files can be compiled through the project window. To compile a file, complete the following steps:

1. Open your project. Select **File** → **Open Project** (on Windows and Motif) or **File** → **Open** (on Macintosh). Select the path and name of your project file.
2. Access the context menu by right clicking over the file you want to compile. Select **Compile** from the context menu. The file is compiled.

For more information on how to compile all the files in your project or just the files that have been recently modified, see [“Compiling an Application from a Project”](#) on page 101.

Note

On Macintosh, you will see a red check mark to the left of each file that has not been compiled after it has been modified.

Testing a File

All IDL GUIBuilder files (.prc) can be run under test mode directly through a project. To run a .prc file in test mode, complete the following steps:

1. Open your project. Select **File** → **Open Project** (on Windows and Motif) or **File** → **Open** (on Macintosh). Select the path and name of your project file.
2. Access the context menu by right clicking over the file you want to test. Select **Test** from the context menu. The file is run in test mode.

For more information on running .prc files in test mode, see “[Running the Application in Test Mode](#)” in Chapter 17 of the *Building IDL Applications* manual.

Setting the Properties of a File

Each file in a project has properties. The following table describes each property of a file:

Property	Description
File name	The name of the file. (This field is read only.)
Path	The path of the file. (This field is read only.)
Group	The name of the group in which the file resides. (This field is read only.)
File Not Found	If the source file cannot be found, you can click this button to display a dialog for finding the path to the file.
Do not Compile	Indicates whether or not to compile the file when running or building. For example, you may have include files for your main program that you do not want compiled. Selecting this check box indicates that you do not want this file compiled. Note - You do not need to set this property for non-source files such as data files, image files, etc. These types of files will be automatically excluded from compilation.
Export	Indicates whether or not to export the file when exporting a project. Some files, such as data files that you need to use when creating your application, are files that you do not want to export. When checked, this file will be exported.

Table 2-2: File Properties

To set the properties for a file, complete the following steps:

1. Open your project. Select **File** → **Open Project** (on Windows and Motif) or **File** → **Open** (on Macintosh). Select the path and name of your project file.
2. Click on the plus sign (on Windows and Motif) or the expand arrow (on Macintosh) to expand the listing of the project files until you see the file you want to change.
3. Select the file by left clicking it.
4. Right click over the file and select **Properties** from the menu. The **File Properties** dialog displays.

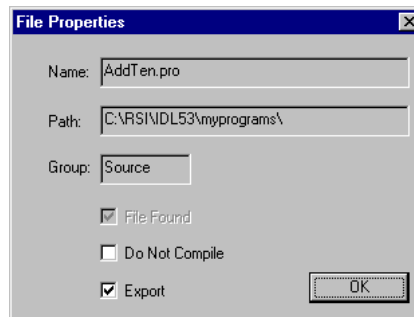


Figure 2-4: File Properties Dialog

5. Change any properties of the file.

Note

On Macintosh, the Do Not Compile option can be selected in the Project Window. If you want the file to be compiled, make sure that the black dot to the right of the file name is displayed. If it is not displayed, click to the right of the file to display it.

6. Click **OK**.
7. Save your project file by selecting **File** → **Save Project**.

Setting the Options for a Project

The options for a project describe how to run, compile, and build a project. To set the options for your project, complete the following steps:

1. Open your project. Select **File** → **Open Project**. Select the path and name of your project file.
2. Click **Project** → **Options...** The **Project Settings** dialog displays.

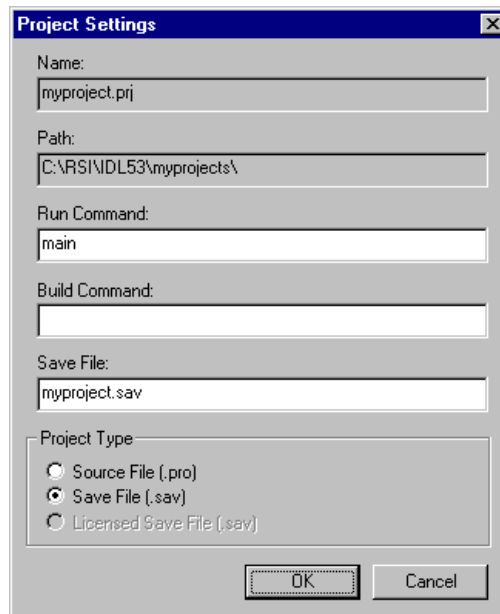


Figure 2-5: Project Settings Dialog

3. Set the options based upon the information in the following table:

Option	Description
Name	Specifies the project name. Note - This field is read only.

Table 2-3: Project Options

Option	Description
Path	<p>Specifies the path of the project.</p> <p>Note - This field is read only.</p>
Run Command	<p>Specifies the IDL command to run your application. The default is the name of the project. This can be any valid IDL command including .sav or .pro files (these can be files that are included or not included in your project.) Typically this is the main program in your application.</p> <p>Tip - You can use the %? command stream substitution to call a dialog to enter a value or values to pass to the called program. For example, if you have a program named “main” and it requires the argument “x” to be passed to it, then you can enter the following for the Run Command:</p> <pre data-bbox="625 683 1176 708">main, %?(Enter the value for x, x)</pre> <p>For more information on how to run your application, see “Running an Application from a Project” on page 104.</p>
Build Command	<p>Specifies the IDL command to build the application. The default is blank. If left blank, the files in the project are built according to the Execution File Format specified and are compiled (if applicable) in the order specified under Build Order. For more information, see “Selecting the Build Order” on page 99.</p> <p>You can enter any valid IDL command including .sav or .pro files. You can also enter a batch file using @filename in order to perform other operations (for example, running a Perl script on your source or data files before compiling. For more information on batch scripts, see the <i>Using IDL</i> manual.</p>
Save File	<p>Specifies the name of the save file to create when building your project. For more information on building a project, see “Building a Project” on page 102.</p> <p>Note - This field is grayed out if you have selected the .pro File Project Type.</p>

Table 2-3: Project Options

Option	Description
Execution File Format	<p>Specifies how the project will run or build. The available formats are:</p> <ul style="list-style-type: none"> • Source File (.pro). • Save File (.sav). • Licensed Save File (.sav) <p>Note - The Licensed Save File option is grayed out if you do not have an IDL Developer Kit license. For more information, see “About IDL Developer’s Kit Licenses” on page 107.</p> <p>For more information on building and running projects, see “Building a Project” on page 102 or “Running an Application from a Project” on page 104.</p>

Table 2-3: Project Options

4. After completing any changes, click **OK**.
5. Save your project file by selecting **File** → **Save Project**.

Note

In addition to setting options for a project, you can also set the properties of a file. For more information, see [“Setting the Properties of a File”](#) on page 94.

Selecting the Build Order

Selecting the build order of a project determines the order in which the files will be compiled. In some cases, you might not be able to run all the files in your project because of dependencies on the order in which they are compiled. For example, if the file `main.pro` contains:

```
Pro main
  x=1
  y=AddTen(x)
  Print, x
End
```

and file `AddTen.pro` contains:

```
Function AddTen, x
  x=x+10
End
```

IDL can't tell if the statement `y=AddTen(x)` is referring to a variable named `AddTen` or a function named `AddTen`. Unless `AddTen` is compiled before `main`, you will get a "Variable undefined" error message.

To select the build order for the files in your project, complete the following steps:

1. Open your project. Select **File** → **Open Project**. Select the path and name of your project file.
2. Click the **Build Order** tab in the Projects window.
3. Move the files to the order in which you want to compile them. The topmost file listed in the Build Order window will be compiled first. On Windows and Macintosh, you can move a file by dragging and dropping it to the desired location. On UNIX, first select a file by left clicking it, then change the order by using the up and down arrows located in the bottom left corner of the Projects window. For example, using the scenario stated previously, the Build Order would look like the following:

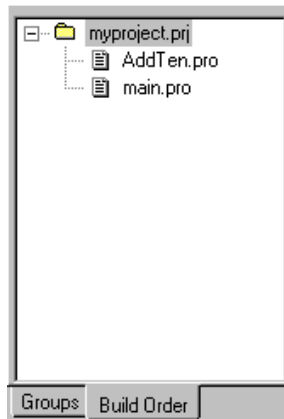


Figure 2-6: Build Order Window

4. Save your project file by selecting **File** → **Save Project**.

Compiling an Application from a Project

You can compile all of your source files or just the files that you have recently modified from your project. A modified file is one that has been modified and then saved (on Macintosh, the file does not have to be saved).

To compile the files in your project, complete the following steps:

1. Open your project. Select **File** → **Open Project**. Select the path and name of your project file.
2. To compile *all the files* in your project on Windows and Motif, select **Project** → **Compile** → **All Files**. On Macintosh, while holding down the Option key, select **Project** → **Compile All Files**.
3. To compile *just the files that have been modified* since the last compilation on Windows and Motif, select **Project** → **Compile** → **Modified Files**. On Macintosh, select **Project** → **Compile Modified Files**.

Note

If you have dependencies on the order in which your files are compiled, see [“Selecting the Build Order”](#) on page 99.

All the files in your project are now compiled. You can now run your application. For more information, see [“Running an Application from a Project”](#) on page 104.

Building a Project

Building a project creates a .sav file of your project or compiles your project based upon the options you have set for your project. If you have specified:

- **Source File** — The IDL session is reset (all procedures, functions, main level variables and common blocks are deleted from memory), all files in the project are compiled, and all undefined but referenced functions and procedures are resolved.

For more information on resetting an IDL session, see [.FULL_RESET_SESSION](#) in the *IDL Reference Guide*. For more information on resolving undefined but referenced functions, see [RESOLVE_ALL](#) in the *IDL Reference Guide*.

- **Save File** — The IDL session is reset (all procedures, functions, main level variables and common blocks are deleted from memory), all files in the project are compiled, all undefined but referenced functions and procedures are resolved, and all the functions and procedures are saved into the file you specified in the project's options.

The save file is created using the XDR and COMPRESS options. For more information, see [SAVE](#) in the *IDL Reference Guide*.

- **Licensed Save File** — The IDL session is reset (all procedures, functions, main level variables and common blocks are deleted from memory), all files in the project are compiled, all undefined but referenced functions and procedures are resolved, all the functions and procedures are saved into the file specified in the project's options, and embedded license information is added to the save file.

For more information on embedded license information, see [“About IDL Developer's Kit Licenses”](#) on page 107.

Note

For more information on project options, see [“Setting the Options for a Project”](#) on page 96.

To build your project, complete the following steps:

1. Open your project. Select **File** → **Open Project**. Select the path and name of your project file.

2. Select **Project** → **Build**. A dialog display confirming that you want to reset your session. This will delete all procedures, functions, main level variables and common blocks from memory so that they will not be saved.
3. Click **OK**.

Your project has been built.

Running an Application from a Project

After compiling your project, you can run your application. What is run depends upon the options you have set for your project:

- If you have selected your execution file format as source file, each file in your project is compiled and then run using the command you specified as the run command.
- If you have selected your execution file format as save file or licensed save file, the most recently compiled version is run using the command you specified as the run command. You must have compiled or built your application before running it.

For more information on setting options for your project, see [“Setting the Options for a Project”](#) on page 96.

To run your application, complete the following steps:

1. Open your project. Select **File** → **Open Project**. Select the path and name of your project file.
2. Select **Project** → **Run**.

Exporting a Project

Exporting a project allows you to create a distribution of your project to either:

- Move a project (including the project file and all of the source files associated with a project to another machine.)
- Create a distribution of your .sav file and an IDL Runtime distribution so that you can distribute it so that others may run your application.

Note

Exporting an IDL Runtime distribution is not supported on Windows NT for Alpha platform.

This is dependent upon the options you have selected for the project. If you have selected:

- **Source File** — Your project’s source, IDL GuiBuilder, data, bitmaps, and any other files listed in your project will be exported to a directory you specify.
- **Save File** — The .sav file for your project will be exported. You will also be given the option of exporting an IDL Runtime distribution for the platform you are exporting on.
- **Licensed Save File** — The .sav file (with an embedded license) for your project will be exported. You will also be given the option of exporting an IDL Runtime distribution for the platform you are exporting on.

For more information on the options for a project, see [“Setting the Options for a Project”](#) on page 96. For more information on creating a .sav file with an embedded license, see [“About IDL Developer’s Kit Licenses”](#) on page 107.

To export your project, complete the following steps:

1. Open your project. Select **File** → **Open Project**. Select the path and name of your project file.
2. If you are exporting a .sav file, you must build the project before exporting. See [“Building a Project”](#) on page 102 for more information.
3. Select **Project** → **Export**. The **Browse for Folder** dialog displays.
4. Select the folder to export the project and click **OK**.

5. If you are exporting a .sav file, a dialog displays asking if you want to export an IDL Runtime distribution with your .sav file. Select **Yes** to include the distribution or **No** to not include the distribution.
6. If you are exporting an IDL Runtime distribution on Windows, enter information on where to copy the distribution files from.

For Windows platforms, you will need to insert your IDL product CD-ROM into your CD-ROM drive. The files needed to create this distribution will be copied from the CD-ROM. Enter the drive letter of your CD-ROM drive that contains your IDL product CD-ROM.

Note

You do not need to specify the path to the IDL distribution for UNIX and Macintosh platforms.

Your project has now been exported.

If you are moving a project from one platform to another, there are a few items to be aware of:

- Project workspace information such as which files are open, etc. will not move from platform to platform.
- Problems with paths can occur if they are not relative paths. If you open a project and find that it cannot find the source file, you can fix this by changing the properties of the file. For more information, see [“Where to Store Source Files for a Project”](#) on page 86 and [“Setting the Properties of a File”](#) on page 94.

For information on how to customize an IDL Runtime distribution and how to distribute it, see [Chapter 3, “Distributing IDL Applications”](#) in the *Building IDL Applications* manual.

About IDL Developer's Kit Licenses

An IDL Developer's Kit License allows a developer to embed licensing information into an IDL application (.sav file). This creates an application that is fully licensed to run on an IDL Runtime distribution. When this embedded license is present, IDL Runtime bypasses normal license checking. One example of this licensing technique is the IDL Demo Applications. The Demo Applications can be run on an unlicensed IDL distribution.

For more information on purchasing a Developer's Kit License, contact your Research Systems sales representative.

Chapter 3:

IDL Development Environment Enhancements

This chapter describes the following topics.

Enhanced Breakpoint Functionality	110	New Color/Font Style Coding for Source Files on UNIX	115
New IDL Functions and Procedures Context Menu for Windows and Motif	114	Enhanced IDL Macros Support	117

Enhanced Breakpoint Functionality

Breakpoints have been enhanced in IDL 5.3. These enhancements include:

- New Tool Bar buttons for easily creating breakpoints, enabling/disabling breakpoints, and displaying the Set Complex Breakpoint dialog.
- You can now selectively create, delete, enable, disable, and set other options from a single dialog (the **Edit Breakpoints** dialog).
- Two new keywords for the BREAKPOINT routine:
 - DISABLE
 - ENABLE
- You can now set breakpoints on a file that has not been compiled. These breakpoints are not enabled until the file is compiled.

The New Breakpoint Tool Bar Buttons

There are now three buttons in the main menu bar. These are:



The **Toggle Breakpoint** button creates or deletes a breakpoint. If you place the cursor in the line you want to create a breakpoint in, clicking the Toggle Breakpoint button creates the breakpoint. If a breakpoint already exists in that line, the breakpoint is removed.



The **Enable/Disable Breakpoint** button enables or disables a breakpoint. If a breakpoint is enabled, a solid circle appears next to the line in the IDL Editor window. If it disabled, the circle is not filled. If a breakpoint has been disabled, the breakpoint is ignored when running the file.



The **Edit Breakpoints** button displays the **Edit Breakpoints** dialog. In previous releases, this printed a listing of the current breakpoints. From this dialog, you can list your current breakpoints, create new breakpoints, enable or disable breakpoints, change breakpoint options, or delete breakpoints.

The New Edit Breakpoints Dialog

The new **Edit Breakpoints** dialog allows you to add, remove, and remove all breakpoints in a file as well as the ability to move to the line in the source file that

contains the breakpoint. The following figure shows the new **Edit Breakpoints** dialog:

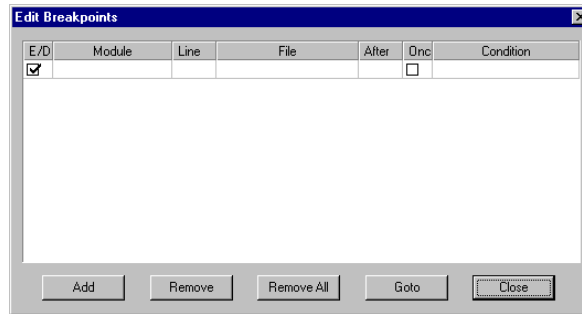



Figure 3-1: Edit Breakpoints Dialog

To create a breakpoint using the new **Edit Breakpoints** dialog, complete the following steps:

1. Open the file you in which you want to set a breakpoint.
2. Display the **Edit Breakpoints** dialog by clicking the  button in the IDLDE Tool Bar or by selecting **Run** → **Edit Breakpoints...**
3. Place the cursor in the line in which you want to create the breakpoint in the Editor window.
4. Select **Add** in the **Edit Breakpoints** dialog box. You will see a new entry display in the dialog. The following table describes each property of a breakpoint:

Item	Description
E/D	Specifies whether a breakpoint is enabled or disabled. If a check mark is displayed, the breakpoint is enabled and execution will stop when the all criteria for the breakpoint is met.
Module	Specifies the procedure or function the breakpoint is set in. Note - This item will not be displayed until the file has been compiled with the new breakpoint.

Table 3-1: Edit Breakpoints Description

Item	Description
Line	Specifies the line number where breakpoint has been set.
File	Specifies the filename where the breakpoint has been set.
After	Specifies how many times the execution must pass the breakpoint before stopping execution. For example, if this item is set to 0, execution will stop the first time this breakpoint is encountered. If it is set to 9, execution will not stop until the breakpoint has been encountered for the ninth time.
Once	The breakpoint is removed after it is encountered for the first time.
Condition	Specifies a condition to be met for the execution to stop. The condition is a string containing an IDL expression. When a breakpoint is encountered, the expression is evaluated. If the expression is true (if it returns a non-zero value), program execution is interrupted. The expression is evaluated in the context of the program containing the breakpoint.

Table 3-1: Edit Breakpoints Description

- At this point, you can modify any of the items (except Module and Line) by double-clicking in the entry.

Your breakpoint entry is now complete.

New Keywords to the BREAKPOINT Routine

The following keywords have been added to the BREAKPOINT routine:

Keyword	Description
DISABLE	Disables the specified breakpoint if it exists. The breakpoint can be specified using the breakpoint index or file and line number.

Table 3-2: New BREAKPOINT Keywords

Keyword	Description
ENABLE	Enables the specified breakpoint if it exists. The breakpoint can be specified using the breakpoint index or file and line number.

Table 3-2: New BREAKPOINT Keywords

New IDL Functions and Procedures Context Menu for Windows and Motif

Previously only available on the Macintosh, the IDL **Function/Procedure Context Menu** has been added to the Windows and UNIX versions of IDL 5.3. The IDL **Function/Procedure Context Menu** allows you to navigate between the different procedures and functions you have defined in the current file you have open in the IDL Editor. On Windows, the menu is located in the main menu bar. On Motif, the menu is located in the upper left corner of the IDL Editor window.

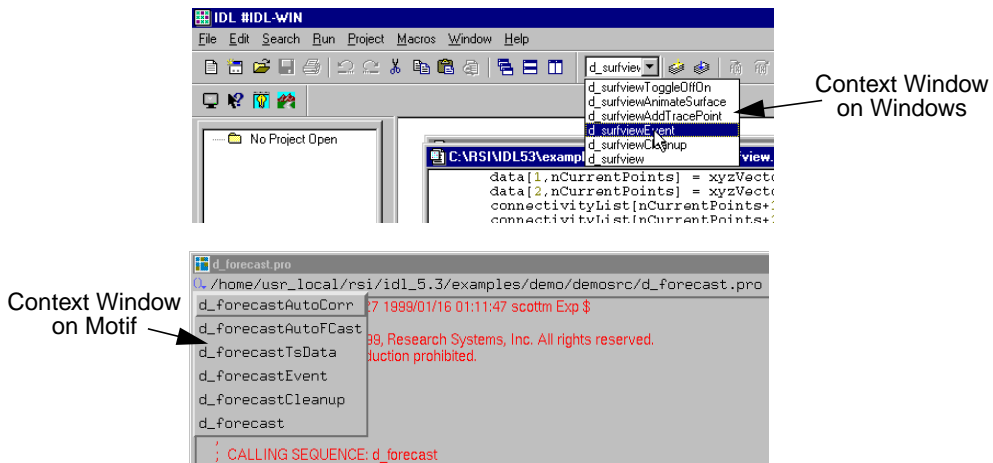


Figure 3-2: IDL Function/Procedure Context Menu on Windows (top) and Motif (bottom)

New Color/Font Style Coding for Source Files on UNIX

Previously available on Windows and Macintosh platforms, color and font style coding has been added to IDL Development Environment for UNIX. This allows you to color code and specify different font styles for the different types of IDL statements that appear in the IDL Editor window.

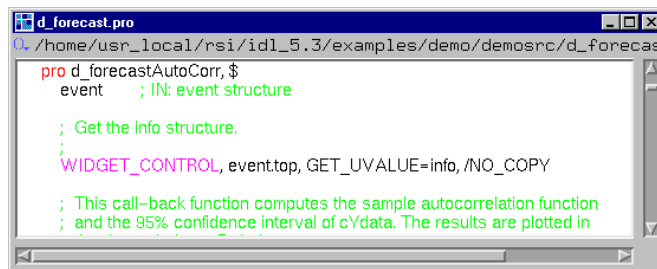


Figure 3-3: Example of Color/Font Coding

To change the color or font, complete the following steps:

1. Select **File** → **Preferences...**
2. On the **Preferences** dialog, click the **Edit** tab.

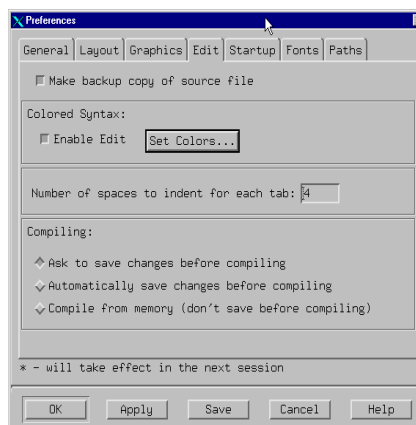


Figure 3-4: Preferences Dialog Edit Tab

3. In the **Colored Syntax** section of the dialog, make sure that the **Enable Edit** button is checked.
4. Click the **Set Colors...** button.
5. The **Set Colors** dialog displays. In this dialog, you can choose the color or font style for a particular statement. Make your selections and click **OK**.

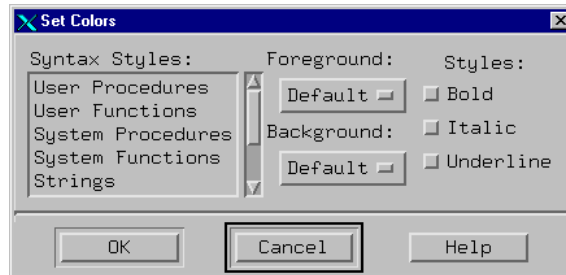


Figure 3-5: Set Colors dialog

You should see the color and font styles you've selected in the current file you have open in the IDL Editor window.

Enhanced IDL Macros Support

Enhancements to IDL macros include:

- New `%?` command stream substitution to display a dialog for input.
- Support for command stream substitutions on the Macintosh.

New `%?` Command Stream Substitution

The `%?` substitution string has been added in IDL 5.3. This substitution string displays a dialog for a user to enter a value to pass to a macro. The syntax for the `%?` substitution string is the following:

```
%?(prompt_text, dialog_title)
```

where *prompt_text* describes the text to prompt the user for entering and *dialog_title* is the text to display in the title bar of the dialog box.

For example, if you want to create a macro that returns the sine of a number, complete the following steps:

1. Select **Macros** → **Edit**.
2. Click the **Add** button.
3. Enter the name of the new macro. For this example, enter “Sine”. Click **OK**.
4. Enter the following in the **IDL Command** field:

```
print, sin(%?(Enter value:, Convert to Sine))
```

5. Enter “Sine” in the **Menu item name** field. Click **OK**.

When you execute the macro by selecting **Macro** → **Sine**, the following dialog displays:

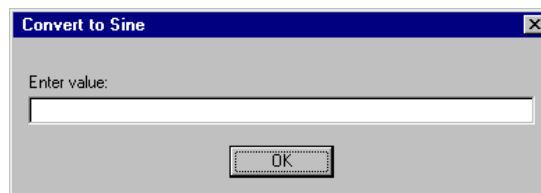


Figure 3-6: `%?` Example Dialog

If you enter a value, the macro will print the sine of that value.

New Support for Command Stream Substitutions on Macintosh

Command stream substitutions are now available on the Macintosh. You can use command stream substitutions to include certain types of information into IDL Macros. The following table lists the available command stream substitutions:

Command	Description
%S	The text of the current selection.
%F or %P	The filename associated with the current IDL Editor.
%N	The base name of the filename (without path and suffix).
%B	The base name of the filename (without path, but with a suffix).
%L	The line number with the current insertion point.
%%	Inserts “%”.
%?	Displays a dialog for entering a value to pass. The syntax is: <p style="text-align: center;"><i>%(prompt_text, dialog_title)</i></p> where <i>prompt_text</i> describes the text to prompt the user for entering and <i>dialog_title</i> is the text to display in the title bar of the dialog box.

Table 3-3: Command Stream Substitutions



Chapter 4: IDL Macros for Importing Data

This chapter describes the following topics.

Overview	120	Using Macros to Import Binary Files	131
Using Macros to Import Image Files	121	Using Macros to Import HDF Files	137
Using Macros to Import ASCII Files	125		

Overview

In IDL 5.3, new macros have been added to ease the importing of data into IDL. This chapter introduces these macros and describes how to import image, ASCII, binary, and Scientific Data Format (SDF) files. These macros are available through the **Macros** menu and also through new IDL **Tool Bar** buttons.

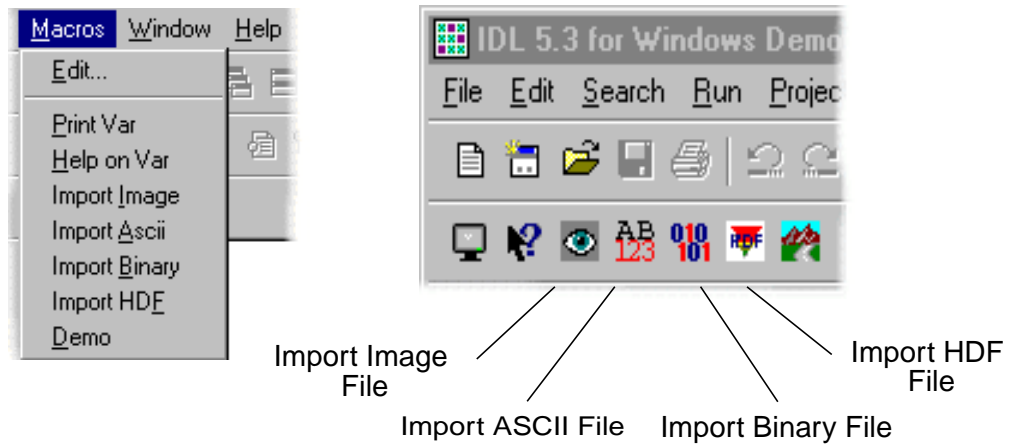


Figure 4-1: Importing Data Macros Menu (Left) and Tool Bar Buttons (Right)

Using Macros to Import Image Files

To import an image file into IDL, complete the following steps:

1. Select the **Import Image File** tool bar button. The **Select Image File** dialog displays.

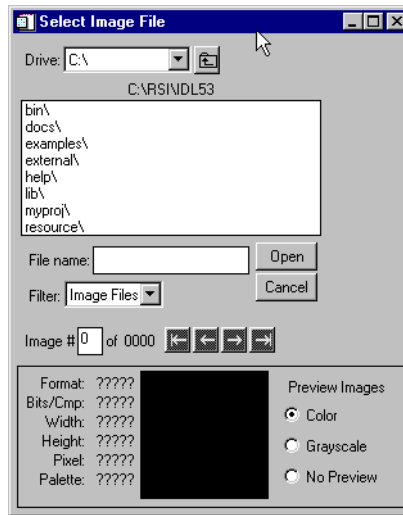


Figure 4-2: Select Image File Dialog

2. Select a file to import. For example, select the *rsi-directory/examples/muscle.jpg* file where *rsi-directory* is the installation directory for IDL.

You can now see a preview of this image as well as other information about the file in the lower section of the Select Image File dialog. You can change the preview to **Color**, **Grayscale**, or **No Preview**. If the image file had more than one actual image, you can see them using the arrow buttons to scroll through the images. You can only read in one image of a multi-image file. The image in the preview is the image that will be read.

3. Click **Open**.
4. The file has been opened into a structure variable named `MUSCLE_IMAGE`.

Images opened with the **Import Image File** macro are stored in structure variables which are named *filename_IMAGE* where *filename* is the name of the file you opened without the extension. So, the file we just opened (muscle.jpg) is now in the structure variable named MUSCLE_IMAGE. The file is a structure as follows:

- IMAGE — The actual image array.
- R — The red color table vectors.
- G — The green color table vectors.
- B — The blue color table vectors.
- QUERY — Contains information about the image.
 - CHANNELS — The number of channels in the image.
 - HAS_PALETTE — Specifies if the palette is present. 1 if the palette is present, else 0. If your image is *n-by-m* the palette is usually present and the R, G, and B color table vectors mentioned above will contain values. If your image is 3-by-*n-by-m*, the palette will not be present and the R,G, and B color table vectors will not contain any values.
 - IMAGE_INDEX — The index of the image of the file. The default is 0, the first image in the file. If there are multiple images in the file that you read, this will be the number (or index) of the image.
 - NUM_IMAGES — The number of images in the original file.
 - PIXEL_TYPE — The IDL **Type Code** of the image pixel format. Valid types are:

PIXEL_TYPE returned	Data Types
1	Byte
2	Integer
3	Longword Integer
4	Floating Point
5	Double-precision Floating Point
12	Unsigned Integer
13	Unsigned Longword Integer

Table 4-1: Values for *PIXEL_TYPE* in the Structure

PIXEL_TYPE returned	Data Types
14	64-bit Integer
15	Unsigned 64-bit Integer

Table 4-1: Values for `PIXEL_TYPE` in the Structure

- `TYPE` — The image type. Valid return values are:
BMP, GIF, JPEG, PNG, PPM, SRF, TIFF, DICOM

The structure can be viewed in the **Variable Watch Window**.

Name	Type	Value
MUSCLE_IMAGE	STRUCT	{ <Anonymous> }
IMAGE	BYTE	Array[652, 444]
R	BYTE	Array[256]
G	BYTE	Array[256]
B	BYTE	Array[256]
QUERY	STRUCT	{ <Anonymous> }
CHANNELS	LONG	1
DIMENSIONS	LONG	Array[2]
HAS_PALETTE	INT	0
IMAGE_INDEX	LONG	0
NUM_IMAGES	LONG	1
PIXEL_TYPE	INT	1
TYPE	STRING	JPEG

Figure 4-3: Variable Watch Window Showing `MUSCLE_IMAGE` Structure

You can specify which part of the structure variable you want to access by using the following syntax:

`variable_name.element_name[.element_name]`

For example, if you want to view the image, enter the following:

```
TV, MUSCLE_IMAGE.IMAGE
```

This displays the following:

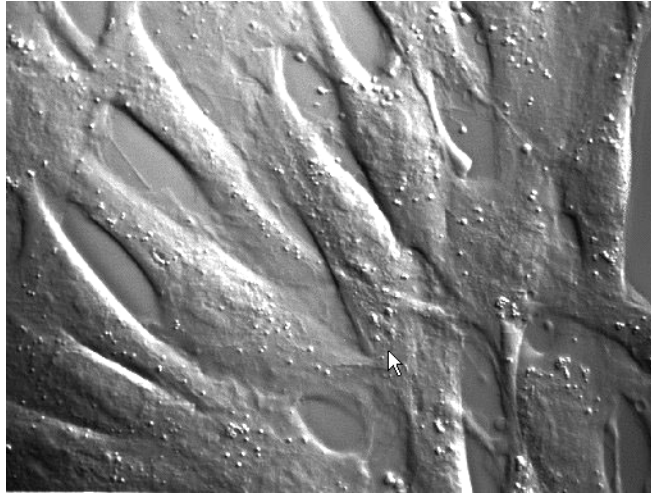


Figure 4-4: MUSCLE_IMAGE.IMAGE

If you want to know the file type, enter the following:

```
PRINT, MUSCLE_IMAGE.QUERY.TYPE
```

IDL prints:

```
JPEG
```

Using Macros to Import ASCII Files

To import an ASCII file into IDL, complete the following steps:

1. Select the **Import ASCII File** tool bar button. The **Select an ASCII file to read** dialog displays.

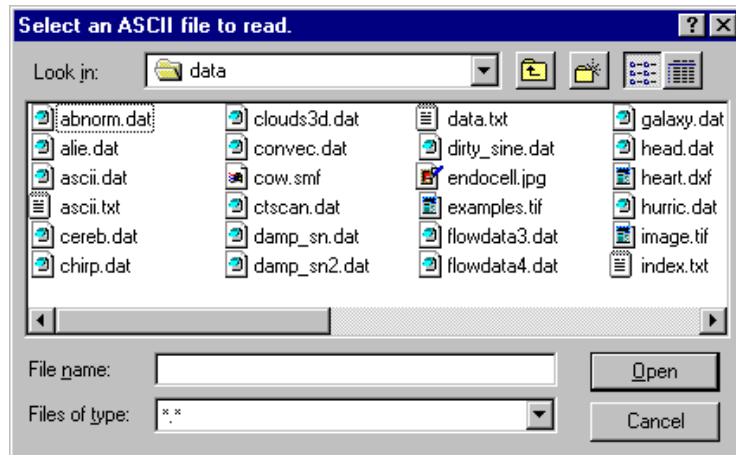


Figure 4-5: Select an ASCII file to read Dialog

2. Select a file to import. For example, select the `rsi-directory/examples/ascii.txt` file where `rsi-directory` is the installation directory for IDL. Click **Open**.
3. In the **Define Data Type/Range** dialog, you specify information about your file. The first few lines of the file are displayed to help you find the information you need to specify.

First, select the type of field which best describes your data. You can either choose **Fixed Width** which specifies that the data is aligned in columns, or **Delimited** which specifies that the data is separated by commas, whitespace, etc. In this example, the data is delimited by commas so we'll select the **Delimited** radio button.

Next, enter a character or string that is used to comment lines within the file in the **Comment String to Ignore:** field. In this example, if we read the first few

lines of this file, it defines the % character as the comment character. Enter the % sign in the **Comment String to Ignore:** field.

Next, enter the line number in which the data starts in the **Data Starts at Line:** field. In this example, the data starts on line 6 so we'll enter that value in the field.

Click **Next**.

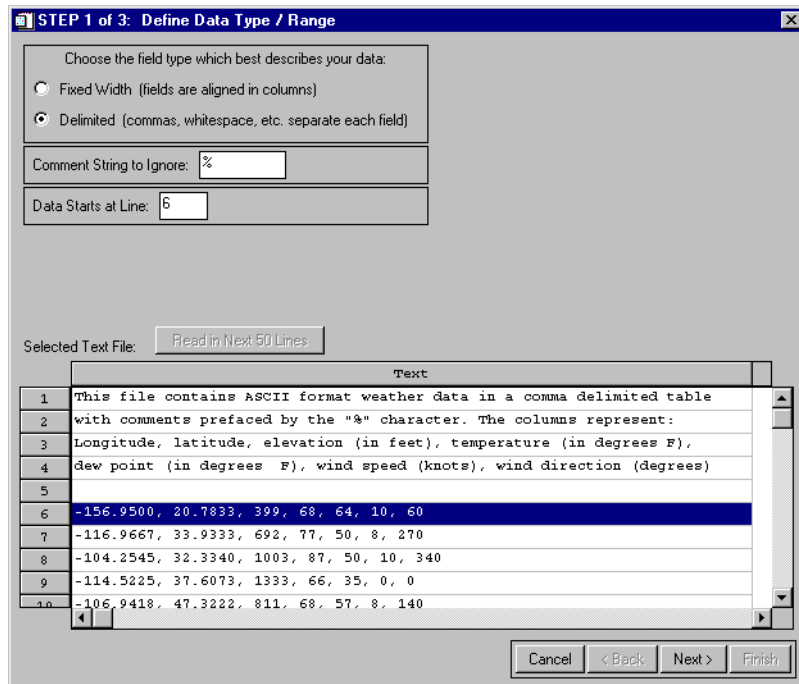


Figure 4-6: Define Data Type/Range Dialog

4. In the **Define Delimiter/Fields** dialog, we'll specify the information about the actual data in the file.

First, we'll enter the number of columns or fields in the **Number of Fields Per Line:** field. In this example, there are 7 fields.

Next, we'll enter the how the data is delimited. You can choose **White Space**, **Comma**, **Colon**, **Semicolon**, **Tab**, or **Other**. If you specify **Other**, you must then enter the characters in the field. In this example, we'll select **Comma** since the data is delimited by commas.

Click **Next**.

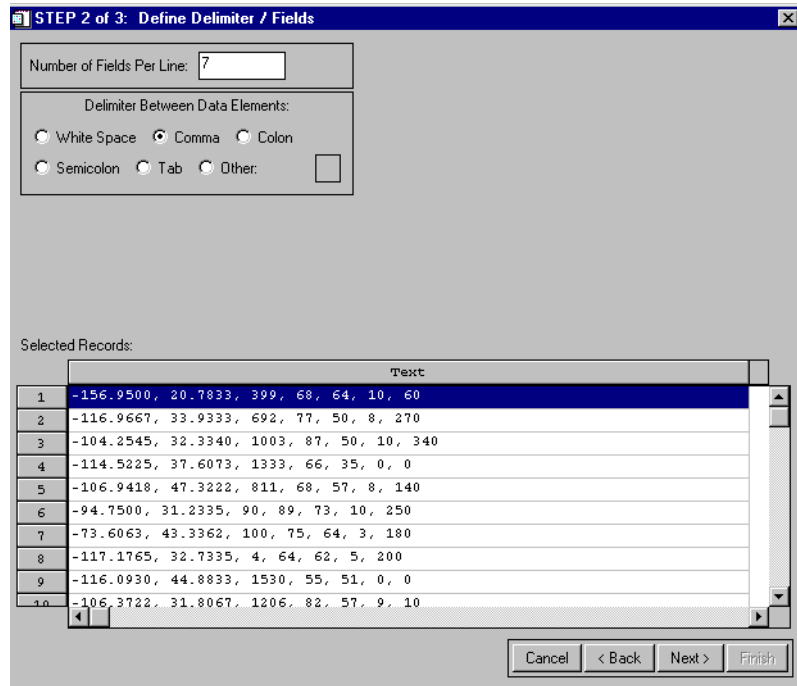


Figure 4-7: Define Delimiter/Fields Dialog

5. In the **Field Specification** dialog, we'll enter information about the contents of each column or field in the data.

First, select the first field in the data in the box in the upper left of the dialog. Enter the name of the field in the **Name** field and the type of data represented in the **Type** field. In this example we'll specify **Longitude** and **Floating** for the this field. Continue naming all the fields in the data using this procedure. In this example, we'll use Latitude – Floating; Elevation – Long; Temperature – Long; DewPoint – Long; WindSpeed – Long; WindDir – Long for the other field pairs.

You can also group some or all of the fields into one field by using the **Group** or **Group All** buttons. In this example, there is no need to group any of the fields.

Next, select the value to assign missing data. You can select the IEEE standard for NaN or a custom value. In this example, we'll choose **IEEE NaN**.

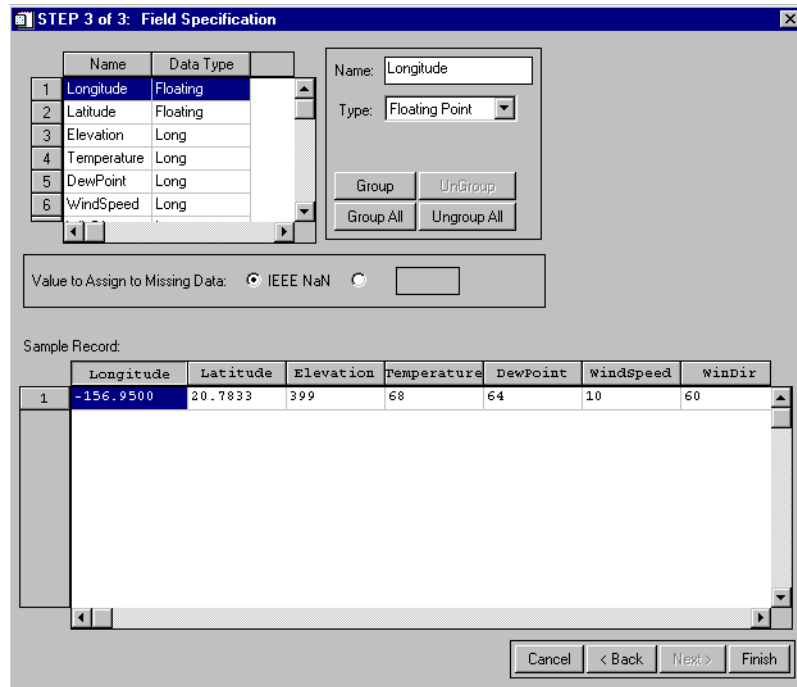


Figure 4-8: Field Specification Dialog

6. Click **Finish**.

ASCII files opened with the **Import ASCII File** macro are stored in structure variables which are named *filename_ASCII* where *filename* is the name of the file you opened without the extension. So, the file we just opened (ascii.txt) is now in the structure variable named ASCII_ASCII. The variable is a structure with each field name being an element of the structure.

The structure can be viewed in the **Variable Watch Window**.

Name	Type	Value
ASCII_ASCII	STRUCT	{ <Anonymous> }
LONGITUDE	FLOAT	Array[15]
LATITUDE	FLOAT	Array[15]
ELEVATION	LONG	Array[15]
TEMPERATURE	LONG	Array[15]
DEWPOINT	LONG	Array[15]
WINDSPEED	LONG	Array[15]
WINDIR	LONG	Array[15]

Figure 4-9: Variable Watch Window Showing ASCII_ASCII Structure

You can specify which part of the structure variable you want to access by using the following syntax:

variable_name.element_name

For example, if you want to view the Longitude field, enter the following:

```
Print, ASCII_ASCII.LONGITUDE
```

IDL prints:

```
-156.950    -116.967    -104.255    -114.522    -106.942
-94.7500    -73.6063    -117.176    -116.093    -106.372
-93.2237    -109.635    -76.0225    -93.1535    -118.721
```

If you want to plot Temperature, enter the following:

```
PLOT, ASCII_ASCII.TEMPERATURE
```

The following figure results.

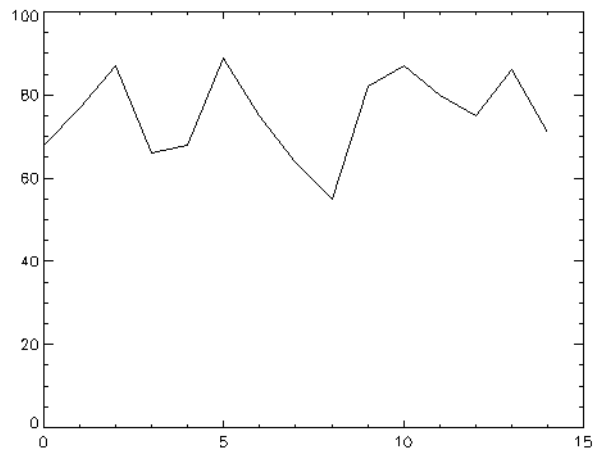


Figure 4-10: Plot of ASCII_ASCII.TEMPERATURE

Using Macros to Import Binary Files

Sometimes, data is stored in files as arrays of bytes instead of a known format like JPEG or TIFF. These files are referred to as binary files.

Note

The **Import Binary File** macro is intended for use in loading raw binary data from files into IDL. Such data is comprised of bits that are meaningful — as integers or floating-point numbers for example — with no special processing (except possibly byte-order swapping) required. Commercial spreadsheet or word processing files, for example, are binary but they are not raw in the above sense, and thus are not good candidates for use with this macro.

Also note that the **Import Binary File** macro is intended for use in loading data from files the contents of which you have some knowledge about. To effectively read data with this macro, you must be able to supply literal values or expressions that specify the type and location of the data in the file you wish to read.

To import a binary file into IDL, complete the following steps:

1. Select the **Import Binary File** tool bar button. The **Select a binary file to read** dialog displays.

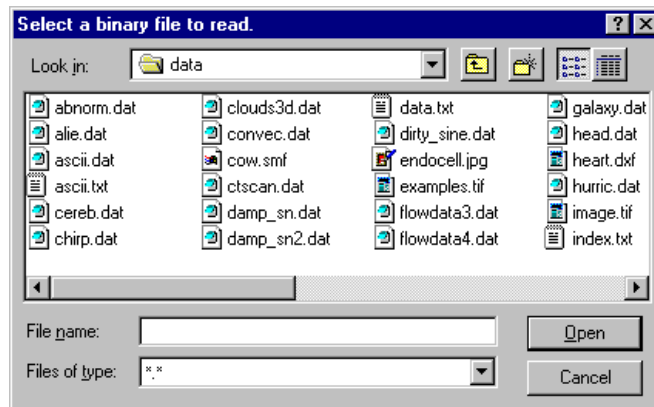


Figure 4-11: Select a binary file to read Dialog

2. Select a file to import. For example, select the `rsi-directory/examples/surface.dat` file where `rsi-directory` is the installation directory for IDL. Click **Open**.
3. In the **Binary Template** dialog box, specify information about your file.

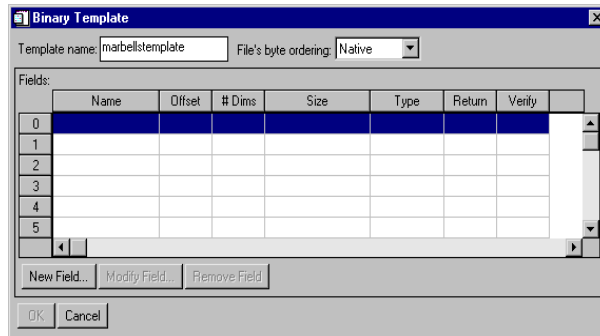


Figure 4-12: The Binary Template dialog

First, enter the name of the template you are going to create in the **Template name:** field. For this example, “marbellstemplate” is used.

Next, select the byte order in the file in the **File’s byte ordering:** pull-down menu. The choices are:

- **Native** — The type of storage method that is native to the machine you are currently running. Little Endian for Intel microprocessor-based machines and Big Endian for Motorola microprocessor-based machines. No byte swapping will be performed.
- **Little Endian** — A method of storing numbers so that the least significant byte appears first in the number. For example, given the hexadecimal number A02B, the little endian method specifies the number to be stored as 2BA0. Specify this if the original file was created on a machine that uses an Intel microprocessor.
- **Big Endian** — A method of storing numbers so that the most significant byte appears first in the number. For example, given the hexadecimal number A02B, the big endian method specifies the number to be stored as A02B. Specify this if the original file was created on a machine that uses a Motorola microprocessor.

The file `surface.dat` was created on a machine that uses an Intel microprocessor. For this example, select **Little Endian** for the byte order.

4. Now we are ready to enter the field values for the file. You can have multiple fields within a binary file. Click the **New Field...** button in the lower-left corner of the **Binary Template** dialog box.

In the **New Field** dialog (shown at the end of these example steps), enter the name of the field in the **Field name:** text box. In this example, enter “A” as the field name.

Next, you need to specify where in the file to start reading. The options are:

- **Offset** — Specifies the byte offset or where to begin reading the file. This is always a decimal integer unless the **Allow an expression for the offset** checkbox is checked. The > symbol specifies to offset forward from a byte position, the < symbol specifies to offset backward from a byte position.
- **From beginning of file** — Specifies to start reading this field starting with the first byte of the file plus any **Offset** specified.
- **From initial position in file/From end of previous field** — This field changes depending upon if this is the first field or any other field besides the first. If this is the first field you are defining, this option specifies to read from the beginning of the file plus any **Offset** specified. If this is not the first field, this option changes to **From end of previous field** and specifies to begin reading the field where the previous field ended plus any **Offset** specified.
- **Allow an expression for the Offset** — If this is checked, you can enter any valid IDL expression in the **Offset** field. You can use any previously defined field in the expression.

In this example, since this is the first field in the file and we don’t have any header information in the file, specify **From the beginning of file** without any offset.

Next, select whether or not you want this field to be returned to IDL when a file is read. For example, you may have a section of your binary file that contains header information. If you create a field for this section, you do not want it returned to IDL. In this case, you would not select **Returned in the result**. You must specify at least one field to be returned to IDL. In this example, we want to return the field we’re creating so we’ll check the box in the upper-right corner marked **Returned in the result**.

Next, you need to specify whether or not you want to verify any of the data you are returning in the **Verified equal to** field. This field is only available if the field is a scalar. This can be any valid IDL expression that evaluates to a scalar. For this example, we won't verify any of the data.

Next, you need to specify the type of data that is in this field. In this example, the data is integer type data so select the Integer (16 bits) at the **Type** pull-down menu. The valid values for **Type** are:

- Byte (unsigned 8-bits)
- Integer (16-bits)
- Long (32-bits)
- Long64 (64 bits)
- Float (32 bits)
- Double-Precision (64-bits)
- Unsigned Integer (16 bits)
- Unsigned Long (32-bits)
- Unsigned Long64 (64-bits)
- Complex (real-imaginary pair of floats)
- Double-Precision Complex (pair of doubles)

Next, specify the number of dimensions contained in the data in the **Number of dimensions:** pull-down menu. This will activate a corresponding number of boxes in the dimensions section of the dialog. In this example, the data is two-dimensional.

Finally, enter the size of each dimension in the field. If you select the **Allow expressions for dimension sizes** check box, you enter any valid IDL expression that returns the size of the dimension. You can also choose to reverse the order of the data by selecting the **Reverse** check box for each dimension. This can be useful when image data is returned in the reverse order and appears upside down. In this example, the data is contained in a 350-by-450 array, so enter 350 for the size of the **1st dimension** and 450 for the size of the **2nd dimension** in the text fields marked **Size:**.

Click **OK**.

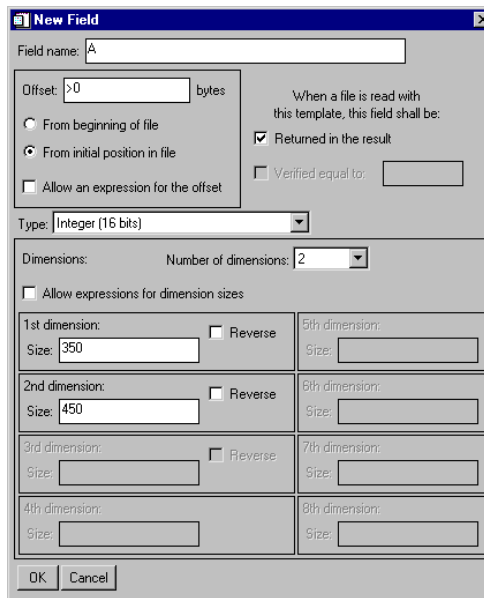


Figure 4-13: Modifying fields in Binary Template

5. You can now see the information that you entered in the **Binary Template** dialog. If you need to enter more fields, select the **New Field** button. Repeat the steps until you have entered all the fields in the binary file.

In this example, there is only one field. Click **OK**.

Binary files opened with the **Import Binary File** macro are stored in structure variables which are named `filename_BINARY` where `filename` is the name of the file you opened without the extension. So, the file we just opened (`surface.dat`) is now in the structure variable named `SURFACE_BINARY`. The variable is a structure with each field name being an element of the structure.

The structure can be viewed in the **Variable Watch Window**.

Name	Type	Value
<input type="checkbox"/> SURFACE_BINARY	STRUCT	{ <Anonymous> }
<input type="checkbox"/> A	INT	Array[350, 450]
<input checked="" type="checkbox"/> [0,0]	INT	3198

Locals Params Common System

Figure 4-14: Variable Watch Window Showing *MARBELLS_BINARY* Structure

You can specify which part of the structure variable you want to access by using the following syntax:

variable_name.element_name

For example, display the image by entering:

```
TVSCL, SURFACE_BINARY.A
```

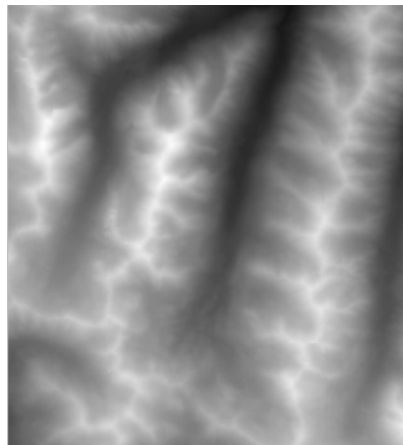


Figure 4-15: *Surface.dat* displayed using TVSCL

Using Macros to Import HDF Files

To import a Hierarchical Data Format (HDF), HDF-EOS, or NETCDF file into IDL, complete the following steps:

1. Select the **Import HDF File** tool bar button. The **Select a valid HDF, NETCDF or HDF-EOS file** dialog is displayed.

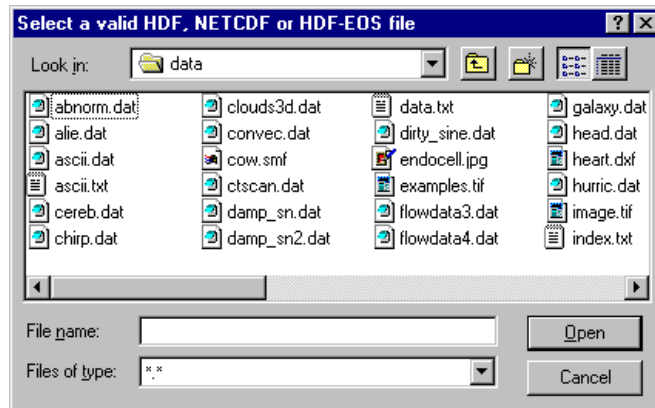


Figure 4-16: Select a valid HDF, NETCDF or HDF-EOS file Dialog

2. Select a file to import. Click **Open**.
3. The **HDF Browser** window is displayed (shown at the end of these example steps). In the **HDF Browser** window, select the data in the file you want to import into IDL.

In the **Display** pull-down menu, select the type of file you are reading. The two options are:

- HDF/NETCDF
- HDF-EOS

Next, select the type of data you want to import. The following tables describe the options available for the two display choices from the pull-down menu.

Menu Selection	Description
HDF/NetCDF Summary	
DF24 (24-bit Images)	24-bit images and their attributes
DFR8 (8-bit Images)	8-bit images and their attributes
DFP (Palettes)	Image palettes
SD (Variables/Attributes)	Scientific Datasets and attributes
AN (Annotations)	Annotations
GR (Generic Raster)	Images
GR Global (File) Attributes	Image attributes
VGroups	Generic data groups
VData	Generic data and attributes

Table 4-2: Menu Options for HDF/NetCDF Data Types

Menu Selection	Description
HDF-EOS Summary	
Point	EOS point data and attributes
Swath	EOS swath data and attributes
Grid	EOS grid data and attributes

Table 4-3: Menu Options for HDF-EOS Data Types

Once you have selected the type of data, information is displayed that shows the different elements of data available in the file you are opening. For example, if it is an image file, you will see the names of the images displayed. Select the item to import.

If you have selected an image, 2D data set, or 3-by- n -by- m data set from the pull-down menu, you can click on the **Preview** button to view the image. If

you have selected a data item that can be plotted in two dimensions, click on the **Preview** button to view a 2D plot of the data (the default); or click on the **Preview Surface** radio button to display a surface plot; click on the **Preview Contour** radio button to display a contour plot; or click on the **Preview Show3** radio button for an image, surface, and contour display. You can also select the **Fit to Window** check box to fit the image to the window.

Next, if you want the data or metadata item you are previewing to be imported into IDL, select the **Read** check box to extract the current data or metadata item from the HDF file.

Next, specify a name for the extracted data or metadata item.

Note

The **Read** check box must be selected for the item to be extracted. Default names are generated for all data items, but may be changed at any time by the user.

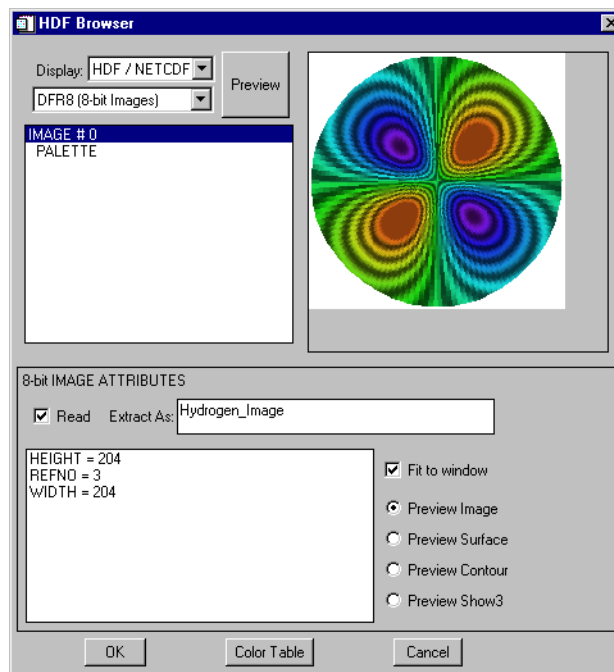


Figure 4-17: HDF Browser Window

4. Continue selecting to read and name the data or metadata items you want to import into IDL.
5. Click **OK**.

HDF, NETCDF, or HDF-EOS files read with the **Import Binary File** macro are stored in structure variables which are named *filename_DF* where *filename* is the name of the file you opened without the extension. The variable is a structure with each data or metadata name being an element of the structure.

You can specify which part of the structure variable you want to access by using the following syntax:

variable_name.data_name

For example, if you imported two data elements out of a file named hydrogen.hdf and you named the elements IMAGE1 and IMAGE2, you could access each individual data element using the following:

HYDROGEN_DF.IMAGE1

HYDROGEN_DF.IMAGE2

If you wanted to view IMAGE1, you would enter:

```
TV, HYDTROGEN_DF . IMAGE1
```

For more information on IDL support of HDF and other scientific data formats, see the *Scientific Data Formats* manual.



Chapter 5: New IDL Routines

This chapter contains documentation for IDL Routines introduced in IDL version 5.3. Complete documentation for IDL Routines (including enhancements to existing routines) can be found in the *IDL Reference Guide*, or in the IDL Online Help. Documentation for the new SDF routines documented in this chapter can also be found in the *Scientific Data Formats* manual.

.FULL_RESET_SESSION

The `.FULL_RESET_SESSION` command does everything `.RESET_SESSION` does, plus the following:

- Removes all system routines installed via `LINKIMAGE` or a `DLM`.
- Removes all structure definitions installed via a `DLM`.
- Removes all message blocks added by `DLMs`.
- Unloads all sharable libraries loaded into IDL via `CALL_EXTERNAL`, `LINKIMAGE`, or a `DLM`.
- Re-initializes all `DLMs` to their unloaded initial state.

Note

The VMS operating system does not support unloading sharable libraries. Therefore, `.FULL_RESET_SESSION` is identical to `.RESET_SESSION` under VMS, and these extra steps are not performed.

Note

`.FULL_RESET_SESSION` is an executive command. Executive commands can only be used at the IDL command prompt, not in programs.

Syntax

```
.FULL_RESET_SESSION
```

.RESET_SESSION

The `.RESET_SESSION` command resets much of the state of an IDL session without requiring the user to exit and restart the IDL session.

`.RESET_SESSION` does the following:

- Returns current execution point to `$MAIN$ (RETALL)`.
- Removes all breakpoints.
- Closes all files except the standard 3 units, the `JOURNAL` file (if any), and any files in use by graphics drivers.
- Destroys/Removes the following:
 - All local variables in `$MAIN$`.
 - All widgets. Exit handlers are not called.
 - All windows and pixmaps for the current window system graphics device are closed. No other graphics state is reset.
 - All common blocks.
 - All handles
 - All user defined system variables
 - All pointer and object reference heap variables.
 - Object destructors are not called.
 - All user defined structure definitions.
 - All user defined object definitions.
 - All compiled user functions and procedures, including the main program (`$MAIN$`), if any.

The following are not reset:

- The current values of intrinsic system variables are retained.
- The saved commands and output log are preserved.
- Graphics drivers are not reset to their full uninitialized state. However, all windows and pixmaps for the current window system device are closed.
- The following files are not closed:

- Stdin (LUN 0)
- Stdout (LUN -1)
- Stderr (LUN -2)
- The journal file (!JOURNAL) if one is open.
- Any files in use by graphics drivers (e.g. PostScript).
- Dynamically loaded graphics drivers (LINKIMAGE) are not removed, nor are any dynamic sharable libraries containing such drivers, even if the same library was also used for another purpose such as CALL_EXTERNAL, LINKIMAGE system routines, or DLMS. See the [.FULL_RESET_SESSION](#) executive command to unload dynamic libraries.

Note

.RESET_SESSION is an executive command. Executive commands can only be used at the IDL command prompt, not in programs.

Syntax

.RESET_SESSION

ADAPT_HIST_EQUAL

The ADAPT_HIST_EQUAL function performs adaptive histogram equalization, a form of automatic image contrast enhancement. The algorithm is described in Pizer et. al., “Adaptive Histogram Equalization and its Variations.”, Computer Vision, Graphics and Image Processing, 39:355-368. Adaptive histogram equalization involves applying contrast enhancement based on the local region surrounding each pixel. Each pixel is mapped to an intensity proportional to its rank within the surrounding neighborhood. This method of automatic contrast enhancement has proven to be broadly applicable to a wide range of images and to have demonstrated effectiveness.

Syntax

```
Result = ADAPT_HIST_EQUAL (Image [, CLIP=value] [, NREGIONS=nregions]
[, TOP=value] )
```

Return Value

The result of the function is a byte image with the same dimensions as the input image parameter.

Arguments

Image

A two-dimensional array representing the image for which adaptive histogram equalization is to be performed. This parameter is interpreted as unsigned 8-bit data, so be sure that the input values are properly scaled into the range of 0 to 255.

Keywords

CLIP

Set this keyword to a nonzero value to clip the histogram by limiting its slope to the given CLIP value, thereby limiting contrast. For example, if CLIP is set to 3, the slope of the histogram is limited to 3. By default, the slope and/or contrast is not limited. Noise over-enhancement in nearly homogeneous regions is reduced by setting this parameter to values larger than 1.0.

NREGIONS

Set this keyword to the size of the overlapped tiles, as a fraction of the largest dimensions of the image size. The default is 12, which makes each tile 1/12 the size of the largest image dimension.

TOP

Set this keyword to the maximum value of the scaled output array. The default is 255.

Example

The following code snippet reads a data file in the IDL Demo data directory containing a cerebral angiogram, and then displays both the original image and the adaptive histogram equalized image:

```
OPENR, 1, DEMO_FILEPATH('cereb.dat', &
  SUBDIRECTORY=['examples', 'data'])

;Image size = 512 x 512
a = BYTARR(512,512, /NOZERO)

;Read it
READU, 1, a
CLOSE, 1

; Reduce size of image for comparison
a = CONGRID(a, 256,256)

;Show original
TVSCL, a, 0

;Show processed
TV, ADAPT_HIST_EQUAL(a, TOP=!D.TABLE_SIZE-1), 1
```

BINARY_TEMPLATE

The BINARY_TEMPLATE function presents a graphical user interface which allows the user to interactively generate a template structure for use with READ_BINARY.

The graphical user interface allows the user to define one or more fields in the binary file. The file may be big, little, or native byte ordering.

Individual fields can be edited by the user to define the dimensionality and type of data to be read. Where necessary, fields can be defined in terms of other previously defined fields using IDL expressions. Fields can also be designated as “Verify”. When a file is read using a template with “Verify” fields, those fields will be checked against a user defined value supplied via the template.

Syntax

```
Template = BINARY_TEMPLATE ([Filename] [, CANCEL=variable]  
[, GROUP=widget_id] [, N_ROWS=rows] [, TEMPLATE=filename] )
```

Arguments

Filename

A scalar string containing the name of a binary file which may be used to test the template. As the user interacts with the BINARY_TEMPLATE graphical user interface, the user’s input will be tested for correctness against the binary data in the file. If *filename* is not specified, a dialog allows the user to choose the file.

Keywords

CANCEL

Set this keyword to a named variable that will contain the byte value 1 if the user clicked the “Cancel” button, or 0 otherwise.

GROUP

The widget ID of an existing widget that serves as “group leader” for the BINARY_TEMPLATE interface. When a group leader is killed, for any reason, all widgets in the group are also destroyed.

N_ROWS

Set this keyword to the number of rows to be visible in the `BINARY_TEMPLATE`'s table of fields.

Note

The `N_ROWS` keyword is analogous to the `WIDGET_TABLE` and the `Y_SCROLL_SIZE` keywords.

TEMPLATE

Use this keyword to specify an initial template for viewing and editing via the `BINARY_TEMPLATE` graphical user interface.

Note

A greater than (“>”) or less than (“<”) symbol can appear in the `BINARY_TEMPLATE`'s “New Field” and the “Modify Field” dialogs where the offset value is displayed. The presence of either symbol indicates that the supplied offset value is “relative” from the end of the previous field or from the initial position in the file. Greater than means offset forward. Less than means offset backward. “>0” and “<0” are synonymous and mean “offset zero bytes”. The user can delete these special symbols (thereby indicating that their corresponding offset value is not “relative”) by typing over them in the “New Field” or “Modify Field” dialogs where the offset value is displayed.

CDF_COMPRESSION

Note

This is a new SDF routine, and is documented in the *Scientific Data Formats* manual.

The CDF_COMPRESSION procedure sets or returns the compression mode for a CDF file and/or variables. Compression parameters should be set before values are written to the CDF file.

Syntax

```
CDF_COMPRESSION, Id [, GET_COMPRESSION=variable]
[, GET_GZIP_LEVEL=variable] [, GET_VAR_COMPRESSION=variable]
[, GET_VAR_GZIP_LEVEL=variable] [, SET_COMPRESSION = {0 | 1 | 2 | 3 | 5}]
[, SET_GZIP_LEVEL=integer{1 to 9}] [, SET_VAR_COMPRESSION = {0 | 1 | 2 |
3 | 5}] [, SET_VAR_GZIP_LEVEL=integer{1 to 9}] [, VARIABLE=variable name
or index] [, /ZVARIABLE]
```

Arguments

Id

The CDF ID of the file being compressed or queried, as returned from a previous call to CDF_OPEN or CDF_CREATE. Note that CDF compression only works for single-file CDF files.

Keywords

GET_COMPRESSION

Set this keyword to a named variable to retrieve the compression type used for the single-file CDF file. Note that individual CDF variables may have compression types different than the one for the rest of the CDF file.

GET_GZIP_LEVEL

Set this keyword to a named variable in which the current GZIP effort level (1-9) for the CDF file is returned. If the compression type for the file is not GZIP (5), then a value of zero is returned.

GET_VAR_COMPRESSION

Set this keyword to a named variable to retrieve the compression type for the variable identified by the VARIABLE keyword.

GET_VAR_GZIP_LEVEL

Set this keyword to a named variable in which the GZIP effort level (1-9) for the variable specified by the VARIABLE keyword is returned. If the compression type for the variable is not GZIP (5), then a value of zero is returned.

SET_COMPRESSION

Set this keyword to the compression type to be used for the single-file CDF file. Note that individual CDF variables may use compression types different than the one for the rest of the CDF file. Valid compression types are:

- 0 = No Compression
- 1 = Run-Length Encoding
- 2 = Huffman
- 3 = Adaptive Huffman
- 5 = GZIP (see the optional GZIP_LEVEL keyword)

SET_GZIP_LEVEL

This keyword is used to indicate the desired effort for the GZIP compression. This effort must be expressed as a scalar in the range (1-9). If GZIP_LEVEL is not specified upon entry, then the default effort level is taken to be 5. If the SET_GZIP_LEVEL keyword is set to a valid value, and the keyword SET_COMPRESSION is not specified, SET_COMPRESSION is set to GZIP (5).

SET_VAR_COMPRESSION

Set this keyword to the compression type for the variable identified by the VARIABLE keyword. If the variable is a zVariable, and is referred to by index in the VARIABLE keyword, then the keyword ZVARIABLE must be set. The desired variable compression should be set before variable data is added with CDF_VARPUT. Valid compression types are:

- 0 = No Compression
- 1 = Run-Length Encoding
- 2 = Huffman

- 3 = Adaptive Huffman
- 5 = GZIP (see the optional GZIP_LEVEL keyword)

SET_VAR_GZIP_LEVEL

Set this keyword to the GZIP effort level (1-9). If the compression type for the variable is not GZIP (5), no action is performed.

VARIABLE

Set this keyword to the name of a variable or a variable index to set the current variable. This keyword is mandatory when querying/setting the compression parameters of an rVariable or zVariable. Note that if VARIABLE is set to the index of a zVARIABLE, the ZVARIABLE keyword must also be set. If ZVARIABLE is not set, the variable is assumed to be an rVariable.

ZVARIABLE

Set this keyword if the current variable is a zVARIABLE and is referred to by index in the VARIABLE keyword. For example:

```
CDF_COMPRESSION, id, VARIABLE=0, /ZVARIABLE,$
  GET_VAR_COMPRESSION=vComp
```

Special Note About Temporary File Location

CDF creates temporary files whenever files/variables are compressed or uncompressed. By default, these files are created in the current directory. UNIX users can set the environment variable CDF_TMP to set the temporary directory explicitly. VMS users can similarly set the logical name CDF\$TMP to an alternate scratch file directory.

Example

```
; Create a CDF file and define the compression.
; Compression only works on Single-File CDFs:
id=CDF_CREATE('demo.cdf',[10,20],/CLOBBER,/SINGLE_FILE)
CDF_COMPRESSION,id,SET_COMPRESSION=1 ; (Run-length encoding)
att_id=CDF_ATTCREATE(id, 'Date',/GLOBAL)
CDF_ATTPUT,id,'Date',att_id,systemtime()

; Change the compression type for the file to GZIP by using
; SET_GZIP_LEVEL:
CDF_COMPRESSION,id,SET_GZIP_LEVEL=7

; Retrieve compression information:
CDF_COMPRESSION,id,GET_GZIP_LEVEL=glevel,GET_COMPRESSION=gcomp
```

```

HELP,glevel,gcomp

; Create and compress an rVariable:
rid=CDF_VARCREATE(id,'rvar0',[1,1],/CDF_FLOAT)
CDF_COMPRESSION,id,SET_VAR_COMPRESSION=2,VARIABLE='rvar0'
CDF_VARPUT,id,'rvar0',findgen(10,20,5)
CDF_COMPRESSION,id,GET_VAR_COMPRESSION=v_comp,VARIABLE=rid,$
GET_VAR_GZIP_LEVEL=v_glevel
HELP,v_comp,v_glevel
; Create and compress a zVariable:
zid=CDF_varcreate(id,'zvar0',[1,1,1],DIM=[10,20,30],/ZVARIABLE,$
/CDF_DOUBLE)

; You can set a compression and check it in the same call:
CDF_COMPRESSION,id,SET_VAR_GZIP_LEVEL=9,VARIABLE=zid,/ZVARIABLE,$
GET_VAR_GZIP_LEVEL=v_gzip
HELP,v_gzip

CDF_VARPUT,id,zid,dindgen(10,20,30),/ZVARIABLE

; File and variable keywords can be combined in the same call
; (Set calls are processed before Get calls)
CDF_COMPRESSION,id,GET_VAR_COMPRESSION=v_comp,VARIABLE='zvar0',$
/ZVARIABLE,SET_COMPRESSION=2,GET_COMPRESSION=file_comp
HELP,file_comp,v_comp

CDF_DELETE,id

```

IDL Output

GLEVEL	LONG	=	7
GCOMP	LONG	=	5
V_COMP	LONG	=	2
V_GLEVEL	LONG	=	0

(Note that V_GLEVEL is 0, since the variable compression is not GZIP.)

V_GZIP	LONG	=	9
FILE_COMP	LONG	=	2
V_COMP	LONG	=	5

COMPILE_OPT

The `COMPILE_OPT` statement allows the author to give the IDL compiler information that changes some of the default rules for compiling the function or procedure within which the `COMPILE_OPT` statement appears.

Research Systems recommends the use of

```
COMPILE_OPT IDL2
```

in all new code intended for use in a reusable library. We further recommend the use of

```
COMPILE_OPT idl2, HIDDEN
```

in all such routines that are not intended to be called directly by regular users (e.g. helper routines that are part of a larger package).

Note

`COMPILE_OPT` is an IDL statement. For information on using statements, see [Chapter 10, “Statements”](#) in *Building IDL Applications*.

Syntax

```
COMPILE_OPT opt1 [, opt2, ..., optn]
```

Arguments

*opt*_{*n*}

This argument can be any of the following:

- **IDL2** — A shorthand way of saying:

```
COMPILE_OPT DEFINT32, STRICTARR
```

- **DEFINT32** — IDL should assume that lexical integer constants default to the 32-bit type rather than the usual default of 16-bit integers. This takes effect from the point where the `COMPILE_OPT` statement appears in the routine being compiled and remains in effect until the end of the routine. The

following table illustrates how the DEFINT32 argument changes the interpretation of integer constants:

Constant	Normal Type	DEFINT32 Type
Without type specifier:		
42	INT	LONG
'2a'x	INT	LONG
42u	UINT	ULONG
'2a'xu	UINT	ULONG
With type specifier:		
0b	BYTE	BYTE
0s	INT	INT
0l	LONG	LONG
42.0	FLOAT	FLOAT
42d	DOUBLE	DOUBLE
42us	UINT	UINT
42ul	ULONG	ULONG
42ll	LONG64	LONG64
42ull	ULONG64	ULONG64

Table 1: Examples of the effect of the DEFINT32 argument

- **HIDDEN** — This routine should not be displayed by HELP, unless the FULL keyword to HELP is used. This directive can be used to hide helper routines that regular IDL users are not interested in seeing.

A side-effect of making a routine hidden is that IDL will not print a “Compile module” message for it when it is compiled from the library to satisfy a call to it. This makes hidden routines appear built-in to the user.

- **OBSOLETE** — If the user has !WARN.OBS_ROUTINES set to True, attempts to compile a call to this routine will generate warning messages that this routine is obsolete. This directive can be used to warn people that there may be better ways to perform the desired task.

- **STRICTARR** — While compiling this routine, IDL will not allow the use of parentheses to index arrays, reserving their use only for functions. Square brackets are then the only way to index arrays. Use of this directive will prevent the addition of a new function in future versions of IDL, or new libraries of IDL code from any source, from changing the meaning of your code, and is an especially good idea for library functions.

Use of STRICTARR can eliminate many uses of the FORWARD_FUNCTION definition.

CW_FILESEL

The CW_FILESEL function is a compound widget for file selection.

Syntax

```
Result = CW_FILESEL (Parent [, /FILENAME] [, FILTER=string array]
[, /FIX_FILTER] [, /FRAME] [, /IMAGE_FILTER] [, /MULTIPLE] [, PATH=string]
[, UNAME=string] [, UVALUE=value] )
```

Arguments

Parent

The widget ID of the parent.

Keywords

FILENAME

Set this keyword to have the initial filename filled in the filename text area.

FILTER

Set this keyword to an array of strings determining the filter types. If not set, the default is “All Files”. All files containing the chosen filter string will be displayed as possible selections. “All Files” is a special filter which returns all files in the current directory.

Example:

```
FILTER=["All Files", ".gif", ".txt"]
```

Multiple filter types may be used per filter entry, using a comma as the separator.

Example:

```
FILTER=[".jpg, .jpeg", ".txt, .text"]
```

FIX_FILTER

If set, the user can not change the file filter.

FRAME

If set, a frame is drawn around the widget.

IMAGE_FILTER

If set, the filter “Image Files” will be added to the end of the list of filters. If set, and `FILTER` is not set, “Image Files” will be the only filter displayed. Valid image files are determined from `QUERY_IMAGE`.

MULTIPLE

If set, the file selection list will allow multiple filenames to be selected. The filename text area will not be editable in this case.

PATH

Set this keyword to the initial path the widget is to start in. The default is the current directory.

UNAME

Set this keyword to a string that can be used to identify the widget in your code. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the `WIDGET_INFO` function with the `FIND_BY_UNAME` keyword. The `UNAME` should be unique to the widget hierarchy because the `FIND_BY_UNAME` keyword returns the ID of the first widget with the specified name.

UVALUE

The ‘user value’ to be assigned to the widget.

CW_LIGHT_EDITOR

The CW_LIGHT_EDITOR function creates a compound widget to edit properties of existing IDLgrLight objects in a view. Lights cannot be added or removed from a view using this widget. However, lights can be “turned off or on” by hiding or showing them (i.e., HIDE property). The returned value of this function is the widget ID of a newly-created light editor.

Syntax

```
Result = CW_LIGHT_EDITOR (Parent [, /DIRECTION_DISABLED]
[, /DRAG_EVENTS] [, FRAME=width] [, /HIDE_DISABLED]
[, LIGHT=objref(s)] [, /LOCATION_DISABLED] [, /TYPE_DISABLED]
[, UVALUE=value] [, XSIZE=pixels] [, YSIZE=pixels] [, XRANGE=vector]
[, YRANGE=vector] [, ZRANGE=vector] )
```

Arguments

Parent

The widget ID of the parent widget for the new light editor.

Keywords

DIRECTION_DISABLED

Set this keyword to make the direction widget portion of the compound widget unchangeable by the user. It will appear insensitive and will not generate an event. The default is to allow this property to be changed.

DRAG_EVENTS

Set this keyword to cause events to be generated continuously while a slider in the compound widget is being dragged or when the mouse cursor is being dragged across the draw widget portion of the compound widget. By default, events are only generated when the mouse comes to rest at its final position and the mouse button is released.

When this keyword is set, a large number of events can be generated. On slower machines, poor performance can result. Therefore, this option should only be used when detailed or truly interactive control is required.

Note

Under Microsoft Windows and Macintosh, sliders do not generate these events, but behave just like regular sliders.

FRAME

The value of this keyword specifies the width of a frame (in pixels) to be drawn around the borders of the widget. Note that this keyword is only a ‘hint’ to the toolkit, and may be ignored in some instances. The default is no frame.

HIDE_DISABLED

Set this keyword to make the hide widget portion of the compound widget unchangeable by the user. It will appear insensitive and will not generate an event. The default is to allow this property to be changed.

LIGHT

Set this keyword to one or more object references to IDLgrLight to edit. This will replace the current set of lights being edited with the list of lights from this keyword.

LOCATION_DISABLED

Set this keyword to make the location widget portion of the compound widget unchangeable by the user. It will appear insensitive and will not generate an event. The default is to allow this property to be changed.

TYPE_DISABLED

Set this keyword to make the light type widget portion of the compound widget unchangeable by the user. It will appear insensitive and will not generate an event. The default is to allow this property to be changed.

UNAME

Set this keyword to a string that can be used to identify the widget. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the WIDGET_INFO function with the FIND_BY_UNAME keyword. The UNAME should be unique to the widget hierarchy because the FIND_BY_UNAME keyword returns the ID of the first widget with the specified name.

UVALUE

The 'user value' to be assigned to the widget. Each widget can contain a user-specified value of any data type and organization. This value is not used by the widget in any way, but exists entirely for the convenience of the IDL programmer. This keyword allows you to set this value when the widget is first created. If UVALUE is not present, the widget's initial user value is undefined.

XRANGE

A two-element vector defining the data range in the x direction. This keyword is used to determine the valid range for the light's location and direction properties

XSIZE

The width of the drawable area in pixels. The default width is 180.

YRANGE

A two-element vector defining the data range in the y direction. This keyword is used to determine the valid range for the light's location and direction properties.

YSIZE

The height of the drawable area in pixels. The default height is 180.

ZRANGE

A two-element vector defining the data range in the z direction. This keyword is used to determine the valid range for the light's location and direction properties

Light Editor Events

There are variations of the light editor event structure depending on the specific event being reported. All of these structures contain the standard three fields (ID, TOP, and HANDLER). The different light editor event structures are described below.

Light Selected

This is the type of structure returned when the light selected in the light list box is modified by a user.

```
{ CW_LIGHT_EDITOR_LS, ID:0L, TOP:0L, HANDLER:0L, LIGHT:OBJ_NEW() }
```

LIGHT specifies the object ID of the new light selection.

Light Modified

This is the type of structure returned when the user has modified a light property. This event maybe generated continuously if the DRAG_EVENTS keyword was set. See DRAG_EVENTS above.

```
{ CW_LIGHT_EDITOR_LM, ID:0L, TOP:0L, HANDLER:0L}
```

The value of the light editor will need to be retrieved (i.e., CW_LIGHT_EDITOR_GET) in order to determine the extent of the actual user modification.

WIDGET_CONTROL Keywords

The widget ID returned by this compound widget is actually the ID of the compound widget's base widget. This means that many keywords to the WIDGET_CONTROL and WIDGET_INFO routines that affect or return information on base widgets can be used with this compound widget (e.g., UNAME, UVALUE).

GET_VALUE

Set this keyword to a named variable to contain the current value of the widget. An IDLgrLight object reference of the currently selected light is returned. The value of a widget can be set with the SET_VALUE keyword to this routine.

SET_VALUE

Sets the value of the specified light editor compound widget. This widget accepts an IDLgrLight object reference of the light in the list of lights to make as the current selection. The property values are retrieved from the light object and the light editor controls are updated to reflect those properties.

CW_LIGHT_EDITOR_GET

The CW_LIGHT_EDITOR_GET procedure gets the CW_LIGHT_EDITOR properties.

Syntax

```
CW_LIGHT_EDITOR_GET, WidgetID [, DIRECTION_DISABLED=variable]
[, DRAG_EVENTS=variable] [, HIDE_DISABLED=variable] [, LIGHT=variable]
[, LOCATION_DISABLED=variable] [, TYPE_DISABLED=variable]
[, XSIZE=variable] [, YSIZE=variable] [, XRANGE=variable]
[, YRANGE=variable][, ZRANGE=variable]
```

Arguments

WidgetID

The widget ID of the CW_LIGHT_EDITOR compound widget.

Keywords

DIRECTION_DISABLED

Set this keyword to a named variable that will contain a boolean value indicating whether this option has been set to make the direction widget portion of the compound widget unchangeable by the user. It will appear insensitive and will not generate an event.

DRAG_EVENTS

Set this keyword to a named variable that will contain a boolean value indicating whether this option has been set to cause events to be generated continuously while a slider in the compound widget is being dragged or when the mouse cursor is being dragged across the draw widget portion of the compound widget.

When this keyword is set, a large number of events can be generated. On slower machines, poor performance can result. Therefore, this option should only be used when detailed or truly interactive control is required.

Note

Under Microsoft Windows and Macintosh, sliders do not generate these events, but behave just like regular sliders.

HIDE_DISABLED

Set this keyword to a named variable that will contain a boolean value indicating whether this option has been set to make the hide widget portion of the compound widget unchangeable by the user.

LIGHT

Set this keyword to a named variable that will contain one or more object references to IDLgrLight.

LOCATION_DISABLED

Set this keyword to a named variable that will contain a boolean value indicating whether this option has been set to make the location widget portion of the compound widget unchangeable by the user.

TYPE_DISABLED

Set this keyword to a named variable that will contain a boolean value indicating whether this option has been set to make the light type widget portion of the compound widget unchangeable by the user.

XRANGE

Set this keyword to a named variable that will contain a two-element vector defining the data range in the x direction.

XSIZE

Set this keyword to a named variable that will contain the width of the drawable area in pixels.

YRANGE

Set this keyword to a named variable that will contain a two-element vector defining the data range in the y direction.

YSIZE

Set this keyword to a named variable that will contain the height of the drawable area in pixels.

ZRANGE

Set this keyword to a named variable that will contain a two-element vector defining the data range in the z direction.

CW_LIGHT_EDITOR_SET

The CW_LIGHT_EDITOR_SET procedure sets the CW_LIGHT_EDITOR properties.

Syntax

```
CW_LIGHT_EDITOR_SET, WidgetID [, /DIRECTION_DISABLED]
[, /DRAG_EVENTS] [, /HIDE_DISABLED] [, LIGHT=objref(s)]
[, /LOCATION_DISABLED] [, /TYPE_DISABLED] [, XSIZE=pixels]
[, YSIZE=pixels] [, XRANGE=vector] [, YRANGE=vector] [, ZRANGE=vector]
```

Arguments

WidgetID

The widget ID of the CW_LIGHT_EDITOR compound widget.

Keywords

DIRECTION_DISABLED

Set this keyword to make the direction widget portion of the compound widget unchangeable by the user. It will appear insensitive and will not generate an event.

DRAG_EVENTS

Set this keyword to cause events to be generated continuously while a slider in the compound widget is being dragged or when the mouse cursor is being dragged across the draw widget portion of the compound widget.

When this keyword is set, a large number of events can be generated. On slower machines, poor performance can result. Therefore, this option should only be used when detailed or truly interactive control is required.

Note

Under Microsoft Windows and Macintosh, sliders do not generate these events, but behave just like regular sliders.

HIDE_DISABLED

Set this keyword to make the hide widget portion of the compound widget unchangeable by the user. It will appear insensitive and will not generate an event.

LIGHT

Set this keyword to one or more object references to IDLgrLight to edit. This will replace the current set of lights being edited with the list of lights from this keyword.

LOCATION_DISABLED

Set this keyword to make the location widget portion of the compound widget unchangeable by the user. It will appear insensitive and will not generate an event.

TYPE_DISABLED

Set this keyword to make the light type widget portion of the compound widget unchangeable by the user. It will appear insensitive and will not generate an event.

XRANGE

A two-element vector defining the data range in the x direction. This keyword is used to determine the valid range for the light's location and direction properties.

XSIZE

The width of the drawable area in pixels.

YRANGE

A two-element vector defining the data range in the y direction. This keyword is used to determine the valid range for the light's location and direction properties.

YSIZE

The height of the drawable area in pixels.

ZRANGE

A two-element vector defining the data range in the z direction. This keyword is used to determine the valid range for the light's location and direction properties.

CW_PALETTE_EDITOR

The CW_PALETTE_EDITOR function creates a compound widget to display and edit color palettes. The palette editor is a base that contains a drawable area to display the color palette, a set of vectors that represent the palette and an optional histogram.

Syntax

```
Result = CW_PALETTE_EDITOR (Parent [, DATA=array] [, FRAME=width]
[, HISTOGRAM=vector] [, /HORIZONTAL] [, SELECTION=[start, end]]
[, UNAME=string] [, UVALUE=value] [, XSIZE=width] [, YSIZE=height]
```

Return Value

The returned value of this function is the widget ID of the newly created palette editor.

Graphics Area Components

Reference Color bar

A gray scale color bar is displayed at the top of the graphics area for reference purposes.

Palette Colorbar

A color bar containing a display of the current palette is displayed below the reference color bar.

Channel and Histogram Display

The palette channel vectors are displayed below the palette colorbar. The Red channel is displayed in red, the Green channel in green, the Blue channel in blue, and the optional Alpha channel in purple. The optional Histogram vector is displayed in Cyan.

An area with a white background represents the current selection, with gray background representing the area outside of the current selection. Yellow drag handles are an additional indicator of the selection endpoints. These selection endpoints represent the range for some editing operations. In addition, cursor X,Y values and channel pixel values at the cursor location are displayed in a status area below the graphics area.

Interactive Capabilities

Color Space

A droplist allows selection of RGB, HSV or HLS color spaces. RGB is the default color space. Note that regardless of the color space in use, the color vectors retrieved with the GET_VALUE keyword to widget control are always in the RGB color space.

Editing Mode

A droplist allows selection of the editing mode. Freehand is the default editing mode.

Unless noted below, editing operations apply only to the channel vectors currently selected for editing. Unless noted below, editing operations apply only to the portion of the vectors within the selection indicators.

In *Freehand* editing mode the user can click and drag in the graphics area to draw a new curve. Editable channel vectors will be modified to use the new curve for that part of the X range within the selection that was drawn in Freehand mode.

In *Line Segment* editing mode a click, drag and release operation defines the start point and end point of a line segment. Editable channel vectors will be modified to use the new curve for that part of the X range within the selection that was drawn in Line Segment mode.

In *Barrel Shift* editing mode click and drag operations in the horizontal direction cause the editable curves to be shifted right or left, with the portion which is shifted off the end of selection area wrapping around to appear on the other side of the selection area. Only the horizontal component of drag movement is used.

In *Slide* editing mode click and drag operations in the horizontal direction cause the editable curves to be shifted right or left. Unlike the Barrel Shift mode, the portion of the curves shifted off the end of the selection area does not wrap around. Only the horizontal component of drag movement is used.

In *Stretch* editing mode click and drag operations in the horizontal direction cause the editable curves to be compressed or expanded. Only the horizontal component of drag movement is used.

A number of buttons provide editing operations which do not require cursor input:

The *Ramp* operation causes the selected part of the editable curves to be replaced with a linear ramp from 0 to 255.

The *Smooth* operation causes the selected part of the editable curves to be smoothed.

The *Posterize* operation causes the selected part of the editable curves to be replaced with a series of steps.

The *Reverse* operation causes the selected part of the editable curves to be reversed in the horizontal direction.

The *Invert* operation causes the selected part of the editable curves to be flipped in the vertical direction.

The *Duplicate* operation causes the selected part of the editable curves to be compressed by 50% and duplicated to produce two contiguous copies of the channel vectors within the initial selection.

The *Load PreDefined* droplist choice leads to additional choices of pre-defined palettes. Loading a pre-defined palette replaces only the selected portion of the editable color channels, respecting of the settings of the selection endpoints and editable checkboxes. This allows loading only a single channel or only a portion of a pre-defined palette.

Channel Display and Edit

A row of checkboxes allows the user to indicate which channels of Red, Green, Blue and the optional Alpha channel should be displayed. A second row of checkboxes allows the user to indicate which channels should be edited by the current editing operation. The checkboxes for the Alpha channel will be sensitive only if an Alpha channel is loaded.

Zoom

Four buttons allow the user to zoom the display of the palette.

The “| |” button zooms to show the current selection.

The “+” button zooms in 50%.

The “-” button zooms out 100%.

The “1:1” button returns the display to the full palette.

Scrolling of the Palette Window

When the palette is zoomed to a scale greater than 1:1 the scroll bar at the bottom of the graphics area can be used to view a different part of the palette.

Arguments

Parent

The widget ID of the parent widget for the new palette editor.

Keywords

DATA

A 3x256 byte array containing the initial color values for Red, Green and Blue channels. The value supplied can also be a 4x256 byte array containing the initial color values and the optional Alpha channel. The value supplied can also be an IDLgrPalette object reference. If an IDLgrPalette object reference is supplied it is used internally and is not destroyed on exit. If an object reference is supplied the ALPHA keyword to the CW_PALETTE_EDITOR_SET routine can be used to supply the data for the optional Alpha channel.

FRAME

The value of this keyword specifies the width of a frame (in pixels) to be drawn around the borders of the widget. Note that this keyword is only a “hint” to the toolkit, and may be ignored in some instances. The default is no frame.

HISTOGRAM

A 256 element byte vector containing the values for the optional histogram curve.

HORIZONTAL

Set this keyword for a horizontal layout for the compound widget. This consists of the controls to the right of the display area. The default is a vertical layout with the controls below the display area.

SELECTION

The selection is a two element vector defining the starting and ending point of the selection region of color indexes. The default is [0,255].

UNAME

Set this keyword to a string that can be used to identify the widget. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the WIDGET_INFO function with the FIND_BY_UNAME keyword. The UNAME should be unique to the widget

hierarchy because the `FIND_BY_UNAME` keyword returns the ID of the first widget with the specified name.

UVALUE

The ‘user value’ to be assigned to the widget. Each widget can contain a user-specified value of any data type and organization. This value is not used by the widget in any way, but exists entirely for the convenience of the IDL programmer. This keyword allows you to set this value when the widget is first created. If `UVALUE` is not present, the widget's initial user value is undefined.

XSIZE

The width of the drawable area in pixels. The default width is 256.

YSIZE

The height of the drawable area in pixels. The default height is 256.

Palette Editor Events

There are variations of the palette editor event structure depending on the specific event being reported. All of these structures contain the standard three fields (`ID`, `TOP`, and `HANDLER`). The different palette editor event structures are described below.

Selection Moved

This is the type of structure returned when one of the vertical bars that define the selection region is moved by a user.

```
{ CW_PALETTE_EDITOR_SM, ID:0L, TOP:0L, HANDLER:0L,
  SELECTION:[ 0, 255 ] }
```

`SELECTION` indicates a two element vector defining the starting and ending point of the selection region of color indexes.

Palette Edited

This is the type of structure returned when the user has modified the color palette.

```
{ CW_PALETTE_EDITOR_PM, ID:0L, TOP:0L, HANDLER:0L }
```

The value of the palette editor will need to be retrieved (i.e., `WIDGET_CONTROL`, `GET_VALUE`) in order to determine the extent of the actual user modification.

WIDGET_CONTROL Keywords for Palette Editor

The widget ID returned by this compound widget is actually the ID of the compound widget's base widget. This means that many keywords to the `WIDGET_CONTROL` and `WIDGET_INFO` routines that affect or return information on base widgets can be used with this compound widget (e.g., `UNAME`, `UVALUE`).

GET_VALUE

Set this keyword to a named variable to contain the current value of the widget. A 3xn (RGB) or 4xn (i.e., RGB and ALPHA) array containing the palette is returned.

The value of a widget can be set with the `SET_VALUE` keyword to this routine.

SET_VALUE

Sets the value of the specified palette editor compound widget. This widget accepts a 3xn (RGB) or 4xn (i.e., RGB and ALPHA) array representing the value of the palette to be set. Another type of argument accepted is an `IDLgrPalette` object reference. If an `IDLgrPalette` object reference is supplied it is used internally and is not destroyed on exit.

CW_PALETTE_EDITOR_GET

The CW_PALETTE_EDITOR_GET procedure gets the CW_PALETTE_EDITOR properties.

Syntax

```
CW_PALETTE_EDITOR_GET, WidgetID [, ALPHA=variable]  
[, HISTOGRAM=variable] )
```

Arguments

WidgetID

The widget ID of the CW_PALETTE_EDITOR compound widget.

Keywords

ALPHA

Set this keyword to a named variable that will contains the optional alpha curve.

HISTOGRAM

Set this keyword to a named variable that will contains the optional histogram curve.

CW_PALETTE_EDITOR_SET

The CW_PALETTE_EDITOR_SET procedure sets the CW_PALETTE_EDITOR properties.

Syntax

```
CW_PALETTE_EDITOR_SET, WidgetID [, ALPHA=byte_vector]  
[, HISTOGRAM=byte_vector] )
```

Arguments

WidgetID

The widget ID of the CW_PALETTE_EDITOR compound widget.

Keywords

ALPHA

A 256 element byte vector that describes the alpha component of the color palette. The alpha value may also be set to the scalar value zero to remove the alpha curve from the display.

HISTOGRAM

The histogram is an vector to be plotted below the color palette. This keyword can be used to display a distribution of color index values to facilitate editing the color palette. The histogram value may also be set to the scalar value zero to remove the histogram curve from the display.

DIALOG_READ_IMAGE

The DIALOG_READ_IMAGE function is a graphical interface used for reading image files.

Syntax

```
Result = DIALOG_READ_IMAGE ([Filename] [, BLUE=variable]  
[, DIALOG_PARENT=widget_id] [, FILE=variable] [, FILTER=string]  
[, /FIX_FILTER] [, GREEN=variable] [, IMAGE=variable] [, PATH=string]  
[, QUERY=variable] [, RED=variable] [, TITLE=string] )
```

Return Value

This routine returns 1 if the “Open” button was clicked, and 0 if the “Cancel” button is clicked.

Arguments

Filename

An optional scalar string containing the full pathname of the file to be highlighted.

Keywords

BLUE

Set this keyword to a named variable that will contain the blue channel vector (if any).

DIALOG_PARENT

The widget ID of a widget that calls DIALOG_READ_IMAGE. When this ID is specified, a death of the caller results in the death of the DIALOG_READ_IMAGE dialog.

FILE

Set this keyword to a named variable that will contain the selected filename with full path when the dialog is created.

FILTER

Set this keyword to a scalar string containing the format type the dialog filter should begin with. The default is “Image Files”. The user cannot modify the filter if the

`FIX_FILTER` keyword is set. Valid values are obtained from the list of supported image types returned from `QUERY_IMAGE`. In addition, there is also the “All Files” type. If set to “All Files”, queries will only happen on filename clicks, making the dialog much more efficient.

Example:

```
FILTER='.jpg, .gif, .tiff'
```

FIX_FILTER

When this keyword is set, only files that satisfy the filter can be selected. The user has no ability to modify the filter.

GREEN

Set this keyword to a named variable that will contain the green channel vector (if any).

IMAGE

Set this keyword to a named variable that will contain the image array read. If Cancel was clicked, no action is taken.

PATH

Set this keyword to a string that contains the initial path from which to select files. If this keyword is not set, the current working directory is used.

QUERY

Set this keyword to a named variable that will return the `QUERY_IMAGE` structure associated with the returned image. If the “Cancel” button was pressed, the variable set to this keyword is not changed. If an error occurred during the read, the `FILENAME` field of the structure will be a null string.

RED

Set this keyword to a named variable that will contain the red channel vector (if any).

TITLE

Set this keyword to a scalar string to be used for the dialog title. If it is not specified, the default title is “Select Image File”.

DIALOG_WRITE_IMAGE

The DIALOG_WRITE_IMAGE function is a graphical user interface used for writing image files.

Syntax

```
Result = DIALOG_WRITE_IMAGE (Image [, R, G, B]  
[, DIALOG_PARENT=widget_id] [, FILENAME=string] [, /FIX_TYPE]  
[, /NOWRITE] [, OPTIONS=variable] [, PATH=string] [, TITLE=string]  
[, TYPE=variable] )
```

Return Value

This routine returns 1 if the “Save” button was clicked, and 0 if the “Cancel” button was clicked.

Arguments

Image

The array to be written to the image file.

R, G, B (optional)

These are optional arguments defining the Red, Green, and Blue color tables to be associated with the image array.

Keywords

DIALOG_PARENT

The widget ID of a widget that calls DIALOG_WRITE_IMAGE. When this ID is specified, a death of the caller results in the death of the DIALOG_WRITE_IMAGE dialog.

FILENAME

Set this keyword to a scalar string that contains the name of the initial file selection. This keyword is useful for specifying a default filename.

FIX_TYPE

When this keyword is set, only files that satisfy the type can be selected. The user has no ability to modify the type.

NOWRITE

Set this keyword to prevent the dialog from writing the file when “Save” is clicked. No data conversions will take place when the save type is chosen.

OPTIONS

Set this keyword to a named variable to contain a structure of the chosen options by the user, including the filename and image type chosen.

PATH

Set this keyword to a string that contains the initial path from which to select files. If this keyword is not set, the current working directory is used.

TITLE

Set this keyword to a scalar string to be used for the dialog title. If it is not specified, the default title is “Save Image File”.

TYPE

Set this keyword to a scalar string containing the format type the “Save as type” field should begin with. The default is “TIFF”. The user can modify the type unless the `FIX_TYPE` keyword is set. Valid values are obtained from the list of supported image types returned from `QUERY_IMAGE`. The “Save as type” field will reflect the type of the selected file (if one is selected).

DLM_LOAD

Normally, IDL system routines that reside in Dynamically Loadable Modules (DLMs) are automatically loaded on demand when a routine from a DLM is called. The DLM_LOAD procedure can be used to explicitly cause a DLM to be loaded.

Syntax

```
DLM_LOAD, DLMNameStr1 [, DLMNameStr2, ..., DLMNameStrn]
```

Arguments

DLMNameStr_n

A string giving the name of the DLM to be loaded. DLM_LOAD causes each named DLM to be immediately loaded.

Keywords

None.

Example

Force the JPEG DLM to be loaded:

```
DLM_LOAD, 'jpeg'
```

IDL Output

```
% Loaded DLM: JPEG.
```

DRAW_ROI

The DRAW_ROI procedure draws a region or group of regions to the current Direct Graphics device. The primitives used to draw each ROI are based on the TYPE property of the given IDLanROI object. The TYPE property selects between points, polylines, and filled polygons.

Syntax

```
DRAW_ROI, oROI [, /LINE_FILL] [, SPACING=value]
```

Graphics Keywords: [, CLIP=[X_0 , Y_0 , X_1 , Y_1]] [, COLOR=value] [, /DATA | , /DEVICE | , /NORMAL] [, LINestyle={0 | 1 | 2 | 3 | 4 | 5}] [, /NOCLIP] [, ORIENTATION=*ccw_degrees_from_horiz*] [, PSYM=*integer*{0 to 10}] [, SYMSIZE=value] [, /T3D] [, THICK=value]

Arguments

oROI

A reference to an IDLanROI object to be drawn.

Keywords

LINE_FILL

Set this keyword to indicate that polygonal regions are to be filled with parallel lines, rather than using the default solid fill. When using a line fill, the thickness, linestyle, orientation, and spacing of the lines may be specified by keywords.

SPACING

The spacing, in centimeters, between the parallel lines used to fill polygons.

Graphics Keywords Accepted

CLIP, COLOR, DATA, DEVICE, LINSTYLE, NOCLIP, NORMAL, ORIENTATION, PSYM, SYMSIZE, T3D, THICK

Example

The following example displays an image and collects data for a region of interest. The resulting ROI is displayed as a filled polygon.

```

PRO roi_ex
; Load and display an image.
img=READ_DICOM(FILEPATH('mr_knee.dcm',SUBDIR=['examples','data']))
TV, img

; Create a polygon region object.
oROI = OBJ_NEW('IDLanROI', TYPE=2)

; Print instructions.
PRINT,'To create a region:'
PRINT,' Left mouse: select points for the region.'
PRINT,' Right mouse: finish the region.'

; Collect first vertex for the region.
CURSOR, xOrig, yOrig, /UP, /DEVICE
oROI->AppendData, xOrig, yOrig
PLOTS, xOrig, yOrig, PSYM=1, /DEVICE

;Continue to collect vertices for region until right mouse button.
x1 = xOrig
y1 = yOrig
while !MOUSE.BUTTON ne 4 do begin
    x0 = x1
    y0 = y1
    CURSOR, x1, y1, /UP, /DEVICE
    PLOTS, [x0,x1], [y0,y1], /DEVICE
    oROI->AppendData, x1, y1
endwhile
PLOTS, [x1,xOrig], [y1,yOrig], /DEVICE

; Draw the the region with a line fill.
DRAW_ROI, oROI, /LINE_FILL, SPACING=0.2, ORIENTATION=45, /DEVICE
END

```

ENABLE_SYSRTN

The `ENABLE_SYSRTN` procedure enables/disables IDL system routines. This procedure is intended for use by runtime and callable IDL applications, and is not generally useful for interactive use.

Syntax

```
ENABLE_SYSRTN [, Routines ] [, /DISABLE] [, /EXCLUSIVE] [, /FUNCTIONS]
```

Arguments

Routines

A string scalar or array giving the names of routines to be enabled or disabled. By default, these are procedures, but this can be changed by setting the `FUNCTIONS` keyword.

Keywords

DISABLE

By default, the Routines are enabled. Setting this keyword causes them to be disabled instead.

EXCLUSIVE

By default, `ENABLE_SYSRTN` does not alter routines not listed in `Routines`. If `EXCLUSIVE` is set, the specified routines are taken to be the only routines that should be enabled or disabled, and all other routines have the opposite action applied.

Therefore, setting `EXCLUSIVE` and not `DISABLE` means that the routines in the `Routines` argument are enabled and all other system routines of the same type (function or procedure) are disabled. Setting `EXCLUSIVE` and `DISABLE` means that all listed routines are disabled and all others are enabled.

FUNCTIONS

Normally, `ROUTINES` specifies the names of procedures. Set the `FUNCTIONS` keyword to manipulate functions instead.

Special Cases

The following is a list of cases in which `ENABLE_SYSRTN` is unable to enable or disable a requested routine. All such attempts are simply ignored without issuing an error, allowing the application to run without error in different IDL environments:

- Attempts to enable/disable non-existent system routines.
- Attempts to enable a system routine disabled due to the mode in which IDL is licensed, as opposed to being disabled via `ENABLE_SYSRTN`, are quietly ignored (e.g. demo mode).
- The routines `CALL_FUNCTION`, `CALL_METHOD`, `CALL_PROCEDURE`, and `EXECUTE` cannot be disabled via `ENABLE_SYSRTN`. However, anything that can be called from them *can* be disabled, so this is not a significant drawback.

Examples

To disable the `PRINT` procedure:

```
ENABLE_SYSRTN, /DISABLE, 'PRINT'
```

To enable the `PRINT` procedure and disable all other procedures:

```
ENABLE_SYSRTN, /EXCLUSIVE, 'PRINT'
```

To ensure all possible functions are enabled:

```
ENABLE_SYSRTN, /DISABLE, /EXCLUSIVE, /FUNCTIONS
```

In the last example, all named functions should be disabled and all other functions should be enabled. Since no *Routines* argument is provided, this means that all routines become enabled.

EOS_GD_QUERY

Note

This is a new SDF routine, and is documented in the *Scientific Data Formats* manual.

The EOS_GD_QUERY function returns information about a specified grid.

Syntax

Result = EOS_GD_QUERY (*Filename*, *GridName*, *Info*)

Return Value

This function returns an integer value of 1 if the file is an HDF file with EOS GRID extensions, and 0 otherwise.

Arguments

Filename

A string containing the name of the file to query.

GridName

A string containing the name of the grid to query.

Info

Returns an anonymous structure containing information about the specified grid. The returned structure contains the following fields:

Field	IDL Data Type	Description
ATTRIBUTES	String array	Array of attribute names
DIMENSION_NAMES	String array	Names of dimensions
DIMENSION_SIZES	Long array	Sizes of dimensions
FIELD_NAMES	String array	Names of fields

Table 5-1: Fields of the Info Structure

Field	IDL Data Type	Description
FIELD_RANKS	Long array	Ranks (dimensions) of fields
FIELD_TYPES	Long array	IDL types of fields
GCTP_PROJECTION	Long	GCTP projection code
GCTP_PROJECTION_PARM	Double array	GCTP projection parameters
GCTP_SPHEROID	Long	GCTP spheroid code
GCTP_ZONE	Long	GCTP zone code (for UTM projection)
IMAGE_LOWRIGHT	Double[2]	Location of lower right corner (meters)
IMAGE_UPLIFT	Double[2]	Location of upper left corner (meters)
IMAGE_X_DIM	Long	Number of columns in grid image
IMAGE_Y_DIM	Long	Number of rows in grid image
NUM_ATTRIBUTES	Long	Number of attributes
NUM_DIMS	Long	Number of dimensions
NUM_IDX_MAPS	Long	Number of indexed dimension mapping entries
NUM_MAPS	Long	Number of dimension mapping entries
NUM_FIELDS	Long	Number of fields
NUM_GEO_FIELDS	Long	Number of geolocation field entries
ORIGIN_CODE	Long	Origin code
PIX_REG_CODE	Long	Pixel registration code

Table 5-1: Fields of the Info Structure

EOS_PT_QUERY

Note

This is a new SDF routine, and is documented in the *Scientific Data Formats* manual.

The EOS_PT_QUERY function returns information about a specified point.

Syntax

Result = EOS_PT_QUERY (*Filename*, *PointName*, *Info*)

Return Value

This function returns an integer value of 1 if the file is an HDF file with EOS POINT extensions, and 0 otherwise.

Arguments

Filename

A string containing the name of the file to query.

PointName

A string containing the name of the point to query.

Info

Returns an anonymous structure containing information about the specified point. The returned structure contains the following fields:

Field	IDL Data Type	Description
ATTRIBUTES	String array	Array of attribute names
NUM_ATTRIBUTES	Long	Number of attributes
NUM_LEVELS	Long	Number of levels

Table 5-2: Fields of the Info Structure

EOS_QUERY

Note

This is a new SDF routine, and is documented in the *Scientific Data Formats* manual.

The EOS_QUERY function returns information about the makeup of an HDF-EOS file.

Syntax

Result = EOS_QUERY (*Filename*, *Info*)

Return Value

This function returns an integer value of 1 if the file is an HDF file with EOS extensions, and 0 otherwise.

Arguments

Filename

A scalar string containing the name of the file to query.

Info

Returns an anonymous structure containing information about the contents of the file. The returned structure contains the following fields:

Field	IDL Data Type	Description
GRID_NAMES	String array	Names of grids
NUM_GRIDS	Long	Number of grids in file
NUM_POINTS	Long	Number of points in file
NUM_SWATHS	Long	Number of swaths in file
POINT_NAMES	String array	Names of points
SWATH_NAMES	String array	Names of swaths

Table 5-3: Fields of the Info Structure

EOS_SW_QUERY

Note

This is a new SDF routine, and is documented in the *Scientific Data Formats* manual.

The EOS_SW_QUERY function returns information about a specified swath.

Syntax

Result = EOS_SW_QUERY (*Filename*, *SwathName*, *Info*)

Return Value

This function returns an integer value of 1 if the file is an HDF file with EOS SWATH extensions, and 0 otherwise.

Arguments

Filename

A string containing the name of the file to be queried.

SwathName

A string containing the name of the swath to be queried.

Info

Returns an anonymous structure containing information about the specified swath. The returned structure contains the following fields:

Field	IDL Data Type	Description
ATTRIBUTES	String array	Array of attribute names
DIMENSION_NAMES	String array	Names of dimensions
DIMENSION_SIZES	Long array	Sizes of dimensions
FIELD_NAMES	String array	Names of fields

Table 5-4: Fields of the Info Structure

Field	IDL Data Type	Description
FIELD_RANKS	Long array	Ranks (dimensions) of fields
FIELD_TYPES	Long array	IDL types of fields
GEO_FIELD_NAMES	String array	Names of geolocation fields
GEO_FIELD_RANKS	Long array	Ranks (dimensions) of geolocation fields
GEO_FIELD_TYPES	Long array	IDL types of geolocation fields
IDX_MAP_NAMES	String array	Names of index maps
IDX_MAP_SIZES	Long array	Sizes of index map arrays
NUM_ATTRIBUTES	Long	Number of attributes
NUM_DIMS	Long	Number of dimensions
NUM_FIELDS	Long	Number of fields
NUM_GEO_FIELDS	Long	Number of geolocation fields
NUM_IDX_MAPS	Long	Number of indexed dimension mapping entries
NUM_MAPS	Long	Number of mapping entries
MAP_INCREMENTS	Long array	Increment of each geolocation relation
MAP_NAMES	String array	Names of maps
MAP_OFFSETS	Long array	Offset of each geolocation relation

Table 5-4: Fields of the Info Structure

GET_DRIVE_LIST

The GET_DRIVE_LIST function returns a string array of the names of valid drives / volumes for the file system (Windows / Macintosh only).

Syntax

Result = GET_DRIVE_LIST()

Return Value

This function returns a string array of the names of valid drives/volumes for the file system.

Arguments

None.

Keywords

None.

GRID_TPS

The GRID_TPS function uses thin plate splines to interpolate a set of values over a regular two dimensional grid, from irregularly sampled data values. Thin plate splines are ideal for modeling functions with complex local distortions, such as warping functions, which are too complex to be fit with polynomials.

Given n points, (x_i, y_i) in the plane, a thin plate spline can be defined as:

$$f(x, y) = a_0 + a_1x + a_2y + \frac{1}{2} \sum_{i=0}^{n-1} b_i r_i^2 \log r_i^2$$

with the constraints:

$$\sum_{i=1}^{n-1} b_i = \sum_{i=1}^{n-1} b_i x_i = \sum_{i=0}^{n-1} b_i y_i = 0$$

where $r_i^2 = (x-x_i)^2 + (y-y_i)^2$. A thin plate spline (TPS) is a smooth function, which implies that it has continuous first partial derivatives. It also grows almost linearly when far away from the points (x_i, y_i) . The TPS surface passes through the original points: $f(x_i, y_i) = z_i$.

Note

GRID_TPS requires at least 7 noncolinear points.

Syntax

```
Interp = GRID_TPS (Xp, Yp, Values [, COEFFICIENTS=variable]
[, NGRID = [nx, ny]] [, START = [x0, y0]] [, DELTA = [dx, dy]] )
```

Return Value

The function returns an array of dimension (nx, ny) of interpolated values. If the values argument is a two-dimensional array, the output array has dimensions (nz, nx, ny) , where nz is the leading dimension of the values array allowing for the interpolation of arbitrarily sized vectors in a single call. Keywords can be used to specify the grid dimensions, size, and location.

Arguments

Xp

A vector of x points.

Yp

A vector of y points, with the same number of elements as the Xp argument.

Values

A vector or two-dimensional array of values to interpolate. If values are a two-dimensional array, the leading dimension is the number of values for which interpolation is performed.

Keywords

COEFFICIENTS

A named variable in which to store the resulting coefficients of the thin plate spline function for the last set of Values. The first N elements, where N is the number of input points, contain the coefficients b_i , in the previous equation. Coefficients with subscripts n , $n+1$, and $n+2$, contain the values of a_0 , a_1 , and a_2 , in the above equation.

DELTA

A two-element array of the distance between grid points (d_x , d_y). If a scalar is passed, the value is used for both dx and dy . The default is the range of the xp and yp arrays divided by $(n_x - 1, n_y - 1)$.

NGRID

A two-element array of the size of the grid to interpolate (n_x , n_y). If a scalar is passed, the value is used for both n_x and n_y . The default value is [25, 25].

START

A two-element array of the location of grid point (x_0 , y_0). If a scalar is passed, the value is used for both x_0 and y_0 . The default is the minimum values in the xp and yp arrays.

References

I. Barrodale, et al, "Note: Warping digital images using thin plate splines", Pattern Recognition, Vol 26, No. 2, pp 375-376, 1993.

M. J. D. Powell, “Tabulation of thin plate splines on a very fine two-dimensional grid”, Report No. DAMTP 1992/NA2, University of Cambridge, Cambridge, U.K. (1992).

Example

The following example creates a set of 25 random values defining a surface on a square, 100 units on a side, starting at the origin. Then, we use `GRID_TPS` to create a regularly gridded surface, with dimensions of 101 by 101 over the square, which is then displayed. The same data set is then interpolated using `TRIGRID`, and the two results are displayed for comparison.

```

;X values
x = RANDOMU(seed, 25) * 100

;Y values
y = RANDOMU(seed, 25) * 100

;Z values
z = RANDOMU(seed, 25) * 10

z1 = GRID_TPS(x, y, z, NGRID=[101, 101], START=[0,0], DELTA=[1,1])

;Show the result
LIVE_SURFACE, z1, TITLE='TPS'

;Grid using TRIGRID
TRIANGULATE, x, y, tr, bounds

z2 = TRIGRID(x, y, z, tr, [1,1], [0,0,100, 100], $
    EXTRAPOLATE=bounds)

;Show triangulated surface
LIVE_SURFACE, z2, TITLE='TRIGRID - Quintic'

```


IMAGE_STATISTICS

The IMAGE_STATISTICS procedure computes sample statistics for a given array of values. An optional mask may be specified to restrict computations to a spatial subset of the input data.

Syntax

```
IMAGE_STATISTICS, Data  
[, /Labeled | [, /Weighted] [, Weight_Sum=variable]] [, /Vector]  
[, LUT=array] [, Mask=array] [, Count=variable] [, Mean=variable]  
[, StdDev=variable] [, Data_Sum=variable] [, Sum_of_Squares=variable]  
[, Minimum=variable] [, Maximum=variable] [, Variance=variable]
```

Arguments

Data

An N -dimensional input data array.

Keywords

COUNT

Set this keyword to a named variable to contain the number of samples that correspond to nonzero values within the mask.

DATA_SUM

Set this keyword to a named variable to contain the sum of the samples that lie within the mask.

LABELED

When set, this keyword indicates values in the mask representing region labels, where each pixel of the mask is set to the index of the region in which that pixel belongs (see the LABEL_REGION function in the *IDL Reference Guide*). If the LABELED keyword is set, each statistic's value is computed for each region index. Thus, a vector containing the results is provided for each statistic with one element per region. By default, this keyword is set to zero, indicating that all samples with a corresponding nonzero mask value are used to form a scalar result for each statistic.

Note

The Labeled keyword cannot be used with either the WEIGHT_SUM or the WEIGHTED keywords.

LUT

Set this keyword to a one-dimensional array. For non-floating point input *Data*, the pixel values are looked up through this table before being used in any of the statistical computations. This allows an integer image array to be calibrated to any user specified intensity range for the sake of calculations. The length of this array must include the range of the input array. This keyword may not be set with floating point input data. When signed input data types are used, they are first cast to the corresponding IDL unsigned type before being used to access this array. For example, the integer value -1 looks up the value 65535 in the LUT array.

MASK

An array of N , or $N-1$ (when the VECTOR keyword is used) dimensions representing the mask array. If the Labeled keyword is set, MASK contains the region indices of each pixel; otherwise statistics are only computed for data values where the MASK array is non-zero.

MAXIMUM

Set this keyword to a named variable to contain the maximum value of the samples that lie within the mask.

MEAN

Set this keyword to a named variable to contain the mean of the samples that lie within the mask.

MINIMUM

Set this keyword to a named variable to contain the minimum value of the samples that lie within the mask.

STDDEV

Set this keyword to a named variable to contain the standard deviation of the samples that lie within the mask.

SUM_OF_SQUARES

Set this keyword to a named variable to contain the sum of the squares of the samples that lie within the mask.

VARIANCE

Set this keyword to a named variable to contain the variance of the samples that lie within the mask.

VECTOR

Set this keyword to specify that the leading dimension of the input array is not to be considered spatial but consists of multiple data values at each pixel location. In this case, the leading dimension is treated as a vector of samples at the spatial location determined by the remainder of the array dimensions.

WEIGHT_SUM

Set the WEIGHT_SUM keyword to a named variable to contain the sum of the weights in the mask.

Note

The WEIGHT_SUM keyword cannot be used if the LABELED keyword is specified.

WEIGHTED

If the WEIGHTED keyword is set, the values in the MASK array are used to weight individual pixels with respect to their count value. If a MASK array is not provided, all pixels are assigned a weight of 1.0.

Note

The WEIGHTED keyword cannot be used if the LABELED keyword is specified.

ISOCONTOUR

The ISOCONTOUR procedure interprets the contouring algorithm found in the IDLgrContour object. The algorithm allows for contouring on arbitrary meshes and returns line or orientated tessellated polygonal output. The interface will also allow secondary data values to be interpolated and returned at the output vertices as well.

Syntax

```
ISOCONTOUR, Values, Outconn, Outverts
[, AUXDATA_IN=array, AUXDATA_OUT=variable] [, C_VALUE=vector]
[, GEOMX=vector] [, GEOMY=vector] [, GEOMZ=vector] [, /FILL]
[, LEVEL_VALUES=variable] [, N_LEVELS=levels] [, /OUTCONN_INDICES]
[, POLYGONS=array of polygon descriptions]
```

Arguments

Values

An input vector or a two-dimensional array specifying the values to be contoured.

Outconn

Output variable to contain the connectivity information of the contour geometry in the form: [n0, i(0, 0), i(0, 1)..., i(0, n0-1), n1, i(1, 0), ...].

Outverts

Output variable to contain the contour vertices.

Keywords

AUXDATA_IN

The auxiliary values to be interpolated at contour vertices. If p is the dimensionality of the auxiliary values, set this argument to a p -by- n array (if the *Values* argument is a vector of length n), or to a p -by- m -by- n array (if the *Values* argument is an m -by- n two-dimensional array).

AUXDATA_OUT

If the AUXDATA_IN keyword was specified, set this keyword to a named output variable to contain the interpolated auxiliary values at the contour vertices. If p is the

dimensionality of the auxiliary values, the output is a p -by- n array of values, where n is the number of vertices in *Outverts*.

C_VALUE

Set this keyword to a vector of values for which contour levels are to be generated. If this keyword is set to 0, contour levels will be evenly sampled across the range of the *Values* argument, using the value of the *N_LEVELS* keyword to determine the number of samples.

FILL

Set this keyword to generate an output connectivity as a set of polygons (*Outconn* is in the form used by the *IDLgrPolygon* *POLYGONS* keyword). The resulting representation is as a set of filled contours. The default is to generate line contours (*Outconn* is in the form used by the *IDLgrPolyline* *POLYLINES* keyword).

GEOMX

Set this keyword to a vector or two-dimensional array specifying the *X* coordinates of the geometry with which the contour values correspond. If *X* is a vector, it must match the number of elements in the *Values* argument, or it must match the first of the two dimensions of the *Values* argument (in which case the *X* coordinates will be repeated for each column of data values).

GEOMY

Set this keyword to a vector or two-dimensional array specifying the *Y* coordinates of the geometry with which the contour values correspond. If *Y* is a vector, it must match the number of elements in the *Values* argument, or it must match the first of the two dimensions of the *Values* argument (in which case the *Y* coordinates will be repeated for each column of data values).

GEOMZ

Set this keyword to a vector or two-dimensional array specifying the *Z* coordinates of the geometry with which the contour values correspond.

If *GEOMZ* is a vector or an array, it must match the number of elements in the *Values* argument.

If *GEOMZ* is not set, the geometry will be derived from the *Values* argument (if it is set to a two-dimensional array). In this case connectivity is implied. The *X* and *Y* coordinates match the row and column indices of the array, and the *Z* coordinates match the data values.

LEVEL_VALUES

Set this keyword to a named output variable to receive a vector of values corresponding to the values used to generate the contours. The length of this vector is equal to the number of contour levels generated.

N_LEVELS

Set this keyword to the number of contour levels to generate. This keyword is ignored if the C_LEVELS keyword is set to a vector, in which case the number of levels is derived from the number of elements in that vector. Set this keyword to 0 to indicate that IDL should compute a default number of levels based on the range of data values. This is the default.

OUTCONN_INDICES

Set this keyword to a named output variable to receive an array of beginning and ending indices of connectivity for each contour level.

The output array is of the form: $[\text{start}_0, \text{end}_0, \text{start}_1, \text{end}_1, \dots, \text{start}_{nc-1}, \text{end}_{nc-1}]$, where nc is the number of contour levels.

POLYGONS

Set this keyword to an array of polygonal descriptions that represents the connectivity information for the data to be contoured (as specified in the *Values* argument). A polygonal description is an integer or long array of the form: $[n, i_0, i_1, \dots, i_{n-1}]$, where n is the number of vertices that define the polygon, and $i_0 \dots i_{n-1}$ are indices into the GEOMX, GEOMY, and GEOMZ keywords that represent the polygonal vertices. To ignore an entry in the POLYGONS array, set the vertex count, n to 0. To end the drawing list, even if additional array space is available, set n to -1 .

ISOSURFACE

The ISOSURFACE procedure algorithm expands on the existing SHADE_VOLUME algorithm. It returns topologically consistent triangles by using oriented tetrahedral decomposition internally. This also allows the algorithm to isosurface any arbitrary tetrahedral mesh. If the user provides an optional auxiliary array, the data in this array is interpolated onto the output vertices and is returned as well. This auxiliary data array is allowed to have more than one value at each vertex. Any size leading dimension is allowed as long as the number of values in the subsequent dimensions matches the number of elements in the input Data array.

Syntax

```
ISOSURFACE, Data, Value, Outverts, Outconn  
[, GEOM_XYZ=array, TETRAHEDRA=array]  
[, AUXDATA_IN=array, AUXDATA_OUT=variable]
```

Arguments

Data

Input three-dimensional array of scalars which are to be contoured.

Value

Input scalar contour value. This value specifies the constant-density surface (also called an iso-surface) to be extracted.

Outverts

Output [3, *n*] array of floating point vertices making up the triangle surfaces.

Outconn

Output array of polygonal connectivity values (see IDLgrPolygon, POLYGONS keyword). If no polygons were extracted, this argument returns the array [-1].

Keywords

AUXDATA_IN

Input array of auxiliary data with trailing dimensions being the number of values in Data.

Note

If AUXDATA_IN is specified then AUXDATA_OUT must also be specified.

AUXDATA_OUT

Set this keyword to a named variable that will contain an output array of auxiliary data sampled at the locations in Outverts.

Note

If AUXDATA_OUT is specified then AUXDATA_IN must also be specified.

GEOM_XYZ

A [3,n] input array of vertex coordinates (one for each value in the Data array). This array is used to define the spatial location of each scalar. If this keyword is omitted, Data must be a three-dimensional array and the scalar locations are assumed to be on a uniform grid.

Note

If GEOM_XYZ is specified then TETRAHEDRA must also be specified if either is to be specified.

TETRAHEDRA

An input array of tetrahedral connectivity values. If this array is not specified, the connectivity is assumed to be a rectilinear grid over the input three-dimensional array. If this keyword is specified, the input data array need not be a three-dimensional array. Each tetrahedron is represented by four values in the connectivity array. Every four values in the array correspond to the vertices of a single tetrahedron.

LOCALE_GET

The LOCALE_GET function returns the current locale (string) of the operating platform.

Syntax

Result = LOCALE_GET()

Arguments

None

Keywords

None

MESH_CLIP

The MESH_CLIP function clips a polygonal mesh to an arbitrary plane in space and returns a polygonal mesh of the remaining portion. An auxiliary array of data may also be passed and clipped. This array can have multiple values for each vertex.

Syntax

```
Result = MESH_CLIP (Plane, Vertsin, Connin, Vertsout, Connout  
[, AUXDATA_IN=array, AUXDATA_OUT=variable] [, CUT_VERTS=variable] )
```

Return Value

The return value is the number of triangles in the returned mesh.

Arguments

Plane

Input four element array describing the equation of the plane to be clipped to. The elements are the coefficients (a, b, c, d) of the equation $ax+by+cz+d=0$.

Vertsin

Input array of polygonal vertices [$3, n$].

Connin

Input polygonal mesh connectivity array.

Vertsout

Output array of polygonal vertices.

Connout

Output polygonal mesh connectivity array.

Keywords

AUXDATA_IN

Input array of auxiliary data. If present, these values are interpolated and returned through AUXDATA_OUT. The trailing array dimension must match the number of vertices in the Vertsin array.

AUXDATA_OUT

Set this keyword to a named variable that will contain an output array of interpolated auxiliary data.

CUT_VERTS

Output array of vertex indices (into Vertsout) of the vertices which are considered to be “on” the clipped surface.

MESH_DECIMATE

The MESH_DECIMATE function reduces the density of geometry while preserving as much of the original data as possible. The classic case is to thin out a polygonal mesh to use fewer polygons while preserving the mesh form. The decimation algorithm removes triangles from the mesh. This is done in such a way as to preserve the mesh edges and to remove roughly planar polygons.

Decimation is a memory and CPU intensive process. Expect the decimation of large models to require large amounts of memory and dozens of seconds to complete. As a reference, a model with approximately 36,000 vertices and 70,000 faces requires 20-30 seconds to decimate to 10% of its original size on a typical NT PC with 64Mb RAM and 333MHz Pentium processor.

If the input polygons are not all triangles, IDL converts the polygons to triangles before decimating. For best results, the polygons should all be convex. Note that if the input polygons are not all triangles, then IDL may return more polygons (as triangles) than were submitted as input, even after decimating a percentage of the polygons. IDL applies the PERCENT_POLYGONS keyword value to the polygon list after converting the list to triangles to approximate the same visual effect of decimating the requested percentage of polygons.

IDL takes steps to deal with input data with a wide variation in magnitude. For example, a troublesome input polygon list may have X and Y values in the 10^1 to 10^2 range, while the Z values may have magnitudes of about 10^{20} . If the results of the decimation are unacceptable, consider scaling the input data so that the magnitudes of the data are closer together.

Syntax

```
Result = MESH_DECIMATE (Verts, Conn, Connout [, /VERTICES]  
[, PERCENT_VERTICES=percent | , PERCENT_POLYGONS=percent ] )
```

Return Value

The return value is the number of triangles in the output connectivity array.

Arguments

Verts

Input array of polygonal vertices [3, *n*].

Conn

Input polygonal mesh connectivity array.

Connout

Output polygonal mesh connectivity array.

Note

Some of the vertices in the Verts array may not be referenced by the Connout array.

Keywords**PERCENT_VERTICES**

Set this keyword to the percent of the original vertices to be returned in the Connout array. It specifies the amount of decimation to perform.

PERCENT_POLYGONS

Set this keyword to the percent of the original polygons to be returned in the Connout array. It specifies the amount of decimation to perform.

Note

PERCENT_VERTICES and PERCENT_POLYGONS are mutually exclusive keywords.

VERTICES

If this keyword is set, the decimation is allowed to add or remove vertices. By default, the output connectivity array is restricted to the set of original input vertices.

MESH_ISSOLID

The MESH_ISSOLID function computes various mesh properties and enables IDL to determine if a mesh encloses space (is a solid). If the mesh can be considered a solid, routines can compute the volume of the mesh.

Syntax

Result = MESH_ISSOLID (*Conn*)

Return Value

Returns 1 if the input mesh fully encloses space (assuming no polygonal interpenetration) or 0 otherwise. A mesh is defined to fully enclose space if each edge in the input mesh appears an even number of times in the mesh.

Note

The input polygonal mesh is assumed to contain only planar, convex polygons.

Arguments

Conn

This is an integer or longword array that represents a series of polygon descriptions. Each polygon description takes the form $[n, i_0, i_1, \dots, i_{n-1}]$, where n is the number of vertices that define the polygon, and $i_0 \dots i_{n-1}$ are indices into the vertex array.

Keywords

None.

MESH_MERGE

The MESH_MERGE function merges two polygonal meshes.

Syntax

```
Result = MESH_MERGE (Verts, Conn, Verts1, Conn1 [, /COMBINE_VERTICES]
[, TOLERANCE=value] )
```

Return Value

The function return value is the number of triangles in the modified polygonal mesh connectivity array.

Arguments

Verts

Input/Output array of polygonal vertices [3, *n*]. These are potentially modified and returned to the user.

Conn

Input/Output polygonal mesh connectivity array. This array is modified and returned to the user.

Verts1

Additional input polygonal vertex array [3, *n*].

Conn1

Additional input polygonal mesh connectivity array.

Keywords

COMBINE_VERTICES

If this keyword is set, the routine will attempt to collapse vertices which are at the same location in space into single vertices. If the expression

$$\max(|x_i - x_{i+1}|, |y_i - y_{i+1}|, |z_i - z_{i+1}|) < tolerance$$

is true, the points (i) and ($i+1$) can be collapsed into a single vertex. The result is returned as a modification of the *Verts* argument.

TOLERANCE

This keyword is used to specify the tolerance value used with the COMBINE_VERTICES keyword. The default value is 0.0.

MESH_NUMTRIANGLES

The MESH_NUMTRIANGLES function computes the number of triangles in a polygonal mesh.

Syntax

Result = MESH_NUMTRIANGLES (*Conn*)

Return Value

Returns the number of triangles in the mesh (a quad is considered two triangles).

Arguments

Conn

Polygonal mesh connectivity array.

Keywords

None.

MESH_SMOOTH

The MESH_SMOOTH function performs spatial smoothing on a polygon mesh. This function smooths a mesh by applying Laplacian smoothing to each vertex, as described by the following formula:

$$\vec{x}_{i_{(n+1)}} = \vec{x}_{i_n} + \frac{\lambda}{M} \sum_{j=0}^M (\vec{x}_{j_n} - \vec{x}_{i_n})$$

where:

\vec{x}_{i_n} is vertex i for iteration n

λ is the smoothing factor

M is the number of vertices that share a common edge with x_{i_n} .

Syntax

```
Result = MESH_SMOOTH (Verts, Conn [, ITERATIONS=value]
[, FIXED_VERTICES=array] [, /FIXED_EDGE_VERTICES] [, LAMBDA=value])
```

Return Value

The output of this function is resulting $[3, n]$ array of modified vertices.

Arguments

Verts

Input array of polygonal vertices $[3, n]$.

Conn

Input polygonal mesh connectivity array.

Keywords

ITERATIONS

Number of iterations to smooth. The default value is 50.

FIXED_VERTICES

Set this keyword to an array of vertex indices which are not to be modified by the smoothing.

FIXED_EDGE_VERTICES

Set this keyword to specify that mesh outer edge vertices are not to be modified by the smoothing.

LAMBDA

Smoothing factor. The default value is 0.05.

MESH_SURFACEAREA

The MESH_SURFACEAREA function computes various mesh properties to determine the mesh surface area, including integration of other properties interpolated on the surface of the mesh.

Syntax

```
Result = MESH_SURFACEAREA ( Verts, Conn [, AUXDATA=array]  
[, MOMENT=variable] )
```

Return Value

Returns the cumulative (weighted) surface area of the polygons in the mesh.

Note

The input polygonal mesh is assumed to contain only planar, convex polygons.

Arguments

Verts

Array of polygonal vertices [3, *n*].

Conn

Polygonal mesh connectivity array.

Keywords

AUXDATA

Array of input auxiliary data (one value per vertex). If present, these values are used to weight a vertex for the purpose of the area computation. The surface area integral will linearly interpolate these values over the surface of each triangle. The default weight is 1.0 which results in the basic polygon area.

MOMENT

If this keyword is present, it will return a three element float vector which corresponds to the first order moments computed with respect to the X, Y and Z axis. The computation is:

$$\vec{m} = \sum_{ntris} a_i \vec{c}_i$$

where a is the (weighted) area of the triangle and c is the centroid of the triangle, thus

$$\vec{m} / sarea$$

yields the (weighted) centroid of the polygon mesh.

MESH_VALIDATE

The MESH_VALIDATE function checks for NaN values in vertices, removes unused vertices, and combines close vertices.

Syntax

```
Result = MESH_VALIDATE ( Verts, Conn [, /REMOVE_NAN]
  [, /PACK_VERTICES] [, /COMBINE_VERTICES] [, TOLERANCE=value] )
```

Return Value

The function return value is the number of triangles in the modified polygonal mesh connectivity array.

Arguments

Verts

Input/Output array of polygonal vertices [3, *n*]. These are potentially modified and returned to the user.

Conn

Input/Output polygonal mesh connectivity array. This array is modified and returned to the user.

Keywords

COMBINE_VERTICES

If this keyword is set, the routine will attempt to collapse vertices which are at the same location in space into single vertices. If the expression

$$\max(|x_i - x_{i+1}|, |y_i - y_{i+1}|, |z_i - z_{i+1}|) < tolerance$$

is true, the points (*i*) and (*i*+1) can be collapsed into a single vertex. The result is returned as a modification of the *Verts* argument.

PACK_VERTICES

If this keyword is set, the Verts input array will be packed to exclude any non-referenced vertices. The result is returned in the Verts argument.

REMOVE_NAN

If this keyword is set, the function will remove any polygons from CONN which reference vertices containing NaN values.

TOLERANCE

This keyword is used to specify the tolerance value used with the COMBINE_VERTS keyword. The default value is 0.0.

MESH_VOLUME

The MESH_VOLUME function computes the volume that the mesh encloses.

Syntax

Result = MESH_VOLUME (*Verts*, *Conn* [, /SIGNED])

Return Value

Returns the volume that the mesh encloses. If the mesh does not enclose space (i.e. MESH_ISSOLID() would return 0), this function returns 0.0.

Note

The input polygonal mesh is assumed to contain only planar, convex polygons.

Arguments

Verts

Array of polygonal vertices [3, *n*].

Conn

Polygonal mesh connectivity array.

Keywords

SIGNED

Set this keyword to compute the signed volume. The sign will be negative for a mesh consisting of inward facing polygons.

MORPH_CLOSE

The MORPH_CLOSE function applies the closing operator to a binary or grayscale image. MORPH_CLOSE is simply a dilation operation followed by an erosion operation. The result of a closing operation is that small holes and gaps within the image are filled, yet the original sizes of the primary foreground features are maintained. The closing operation is an idempotent operator, applying it more than once produces no further effect.

Both the opening and the closing operators have the effect of smoothing the image, with the opening operation removing pixels, and the closing operation adding pixels.

Syntax

```
Result = MORPH_CLOSE (Image, Structure [, /GRAY]  
[, PRESERVE_TYPE=bytearray | /UINT | /ULONG] [, VALUES=array] )
```

Arguments

Image

A one-, two-, or three-dimensional array upon which the closing operation is to be performed. If neither of the keywords GRAY or VALUES is present, the image is treated as a binary image with all nonzero pixels considered as 1.

Structure

A one-, two-, or three-dimensional array to be used as the structuring element. The elements are interpreted as binary values - either zero or nonzero. The structuring element must have the same number of dimensions as the *Image* argument.

Keywords

GRAY

Set this keyword to perform a grayscale, rather than binary, operation. Nonzero elements of the *Structure* parameter determine the shape of the structuring element. If the VALUES keyword is not present, all elements of the structuring element are 0.

PRESERVE_TYPE

Set this keyword to return the same type as the input array. The input array must be of type BYTE, UINT, or ULONG. This keyword only applies for grayscale erosion/dilation, and is mutually exclusive of UINT and ULONG.

UINT

Set this keyword to return an unsigned integer array. This keyword only applies for grayscale operations, and is mutually exclusive of the ULONG and PRESERVE_TYPE keywords.

ULONG

Set this keyword to return an unsigned longword integer array. This keyword only applies for grayscale operations, and is mutually exclusive of the UINT and PRESERVE_TYPE keywords.

VALUES

An array of the same dimensions as *Structure* providing the values of the structuring element. The presence of this keyword implies a grayscale operation.

Example

The following code snippet reads a data file in the IDL Demo data directory containing a magnified image of grains of pollen. It then applies a threshold and a morphological closing operator with a 3 by 3 square kernel to the original image. Notice that most of the holes in the pollen grains have been filled by the closing operator.

```

;Read the image
READ_JPEG, DEMO_FILEPATH('pollens.jpg', $
    SUBDIR=['examples', 'demo', 'demodata']), a

;Apply the threshold creating a binary image
b = a ge 140b

;Load a simple color table
TEK_COLOR
TV, b, 0

;Apply closing operator
c = MORPH_CLOSE(b, REPLICATE(1,3,3))

;Show the result
TV, c, 1

;Show added pixels in white
TV, b + c, 2

```

MORPH_DISTANCE

The MORPH_DISTANCE function estimates N -dimensional distance maps, which contain for each foreground pixel the distance to the nearest background pixel, using a given norm. Available norms include: Euclidean, which is exact and is also known as the Euclidean Distance Map (EDM), and two more efficient approximations, chessboard and city block.

The distance map is useful for a variety of morphological operations: thinning, erosion and dilation by discs of radius “ r ”, and granulometry.

Syntax

```
Result = MORPH_DISTANCE (Data [, /BACKGROUND]  
[, NEIGHBOR_SAMPLING={1 | 2 | 3}] [, /NO_COPY])
```

Return Value

The returned variable is an array of the same dimension as the input array.

Arguments

Data

An input binary array. Zero-valued pixels are considered to be part of the background.

Keywords

BACKGROUND

By default, the EDM is computed for the foreground (non-zero) features in the *Data* argument. Set this keyword to compute the EDM of the background features instead of the foreground features.

NEIGHBOR_SAMPLING

Set this keyword to indicate how the distance of each neighbor from a given pixel is determined. The following table describes the valid values:

Setting	Action Taken
0 - default	No diagonal neighbors. Each neighbor is assigned a distance of 1.
1 - chessboard	Each neighbor is assigned a distance of 1.
2 - city block	Each neighbor is assigned a distance corresponding to the number of pixels to be visited when travelling from the current pixel to the neighbor. (The path can only take 90 degree turns; no diagonal paths are allowed.)
3 - actual distance	Each neighbor is assigned its actual distance from the current pixel (within the limitations of floating point representations).

Table 5-5: NEIGHBOR_SAMPLING Settings

Default Two Dimensional Example

```

      1
    1  X  1
      1
  
```

Chessboard Two-Dimensional Example

```

    1  1  1
    1  X  1
    1  1  1
  
```

City Block Two-Dimensional Example:

```

    2  1  2
    1  X  1
    2  1  2
  
```

Actual Distance Two-Dimensional Example

```

sqrt(2)  1  sqrt(2)
    1      X      1
sqrt(2)  1  sqrt(2)
  
```

NO_COPY

Set this keyword to request that the input array be reused, if possible. If this keyword is set, the input argument is undefined upon return.

Example

The following code snippet reads a data file in the IDL Demo data directory containing a magnified image of grains of pollen. It then applies a threshold and the morphological distance operator. Thresholding the result distance operator with a value of “n”, produces the equivalent of eroding the thresholded image with a disc of radius “n”.

```
;Read the image
READ_JPEG, '/usr/local/rsi/idl/examples/demo/demodata/pollens*', a

;Apply the threshold
b = a ge 140b

;Show thresholded image
TVSCL, b, 0

;Create Euclidean distance function
c = MORPH_DISTANCE(b, NEIGHBOR_SAMPLING = 3)

;Show distance function
TVSCL, c, 1

;Show image after erosion with a disc of radius 5
TVSCL, c GT 5, 2
```

MORPH_GRADIENT

The MORPH_GRADIENT function applies the morphological gradient operator to a grayscale image. MORPH_GRADIENT is the subtraction of an eroded version of the original image from a dilated version of the original image. The practical result of a morphological gradient operation is that the boundaries of features are highlighted.

Syntax

```
Result = MORPH_GRADIENT (Image, Structure [, PRESERVE_TYPE=bytearray |  
/UINT | /ULONG] [, VALUES=array] )
```

Arguments

Image

A one-, two-, or three-dimensional array upon which the morphological gradient operation is to be performed.

Structure

A one-, two-, or three-dimensional array to be used as the structuring element. The elements are interpreted as binary values - either zero or nonzero. The structuring element must have the same number of dimensions as the *Image* argument.

Keywords

PRESERVE_TYPE

Set this keyword to return the same type as the input array. The input array must be of type BYTE, UINT, or ULONG. This keyword only applies for grayscale erosion/dilation, and is mutually exclusive of the UINT and ULONG keywords.

UINT

Set this keyword to return an unsigned integer array. This keyword is mutually exclusive of the ULONG and PRESERVE_TYPE keywords.

ULONG

Set this keyword to return an unsigned longword integer array. This keyword is mutually exclusive of the UINT and PRESERVE_TYPE keywords.

VALUES

An array of the same dimensions as the *Structure* argument providing the values of the structuring element. If the VALUES keyword is not present, all elements of the structuring element are 0.

Example

The following code snippet reads a data file in the IDL Demo data directory containing a magnified image of grains of pollen. It then creates disc of radius 2, in a 5 by 5 array, with all elements within a radius of 2 from the center set to 1. This disc is used as the structuring element for the morphological gradient which is then displayed as both a gray scale image, and as a thresholded image.

```
;Radius of disc
r = 2

;Read the image
READ_JPEG, '/usr/local/rsi/idl/examples/demo/demodata/pollens*', a

;Show original image
TVSCL, a, 0

;Create a binary disc of given radius.
disc = SHIFT(DIST(2*r+1), r, r) LE r

b = MORPH_GRADIENT(a, disc)

;Show edges
TVSCL, b, 1

;Show thresholded edges
TVSCL, b ge 100, 2
```

MORPH_HITORMISS

The MORPH_HITORMISS function applies the hit-or-miss operator to a binary image. The hit-or-miss operator is implemented by first applying an erosion operator with a *hit* structuring element to the original image. Then an erosion operator is applied to the complement of the original image with a secondary *miss* structuring element. The result is the intersection of the two results.

The resulting image corresponds to the positions where the hit structuring element lies within the image, and the miss structure lies completely outside the image. The two structures must not overlap.

Syntax

Result = MORPH_HITORMISS (*Image*, *HitStructure*, *MissStructure*)

Arguments

Image

A one-, two-, or three-dimensional array upon which the morphological operation is to be performed. The image is treated as a binary image with all nonzero pixels considered as 1.

HitStructure

A one-, two-, or three-dimensional array to be used as the hit structuring element. The elements are interpreted as binary values — either zero or nonzero. This structuring element must have the same number of dimensions as the *Image* argument.

MissStructure

A one-, two-, or three-dimensional array to be used as the miss structuring element. The elements are interpreted as binary values — either zero or nonzero. This structuring element must have the same number of dimensions as the *Image* argument.

Note

It is assumed that the HitStructure and the MissStructure arguments are disjoint.

Keywords

None.

Example

The following code snippet identifies blobs with a radius of at least 2, but less than 4 in the pollen image. These regions totally enclose a disc of radius 2, contained in the 5 x 5 kernel named “hit”, and in turn, fit within a hole of radius 4, contained in the 9 x 9 array named “miss”. Executing this specific example identifies four blobs in the image with these attributes.

```

;Radius of hit disc
rh = 2

;Radius of miss disc
rm = 4

;Create a binary disc of given radius.
hit = SHIFT(DIST(2*rh+1), rh, rh) LE rh

;Complement of disc for miss
miss = SHIFT(DIST(2*rm+1), rm, rm) GT rm

;Load discrete color table
TEK_COLOR

;Read the image
READ_JPEG, DEMO_FILEPATH('pollens.jpg', $
    SUBDIR=['examples', 'demo', 'demodata']), a

;Apply the threshold
b = a GE 140b

;Show thresholded image
TV, b, 0

;Compute matches
c = MORPH_HITORMISS(b, hit, miss)

;Expand matches to size of hit disc
c = DILATE(c, hit)

;Show matches.
TV, c, 1

;Superimpose, showing hit regions in blue.
;(Blue = color index 4 for tek_color.)
TV, b + 3*c, 2

```

MORPH_OPEN

The MORPH_OPEN function applies the opening operator to a binary or grayscale image. MORPH_OPEN is simply an erosion operation followed by a dilation operation. The result of an opening operation is that small features (e.g., noise) within the image are removed, yet the original sizes of the primary foreground features are maintained. The opening operation is an idempotent operator, applying it more than once produces no further effect.

An alternative definition of the opening, is that it is the union of all sets containing the structuring element in the original image. Both the opening and the closing operators have the effect of smoothing the image, with the opening operation removing pixels, and the closing operation adding pixels.

Syntax

```
Result = MORPH_OPEN (Image, Structure [, /GRAY]  
[, PRESERVE_TYPE=bytearray | /UINT | /ULONG] [, VALUES=array] )
```

Arguments

Image

A one-, two-, or three-dimensional array upon which the opening operation is to be performed. If neither of the keywords GRAY or VALUES is present, the image is treated as a binary image with all nonzero pixels considered as 1.

Structure

A one-, two-, or three-dimensional array to be used as the structuring element. The elements are interpreted as binary values — either zero or nonzero. The structuring element must have the same number of dimensions as the *Image* argument.

Keywords

GRAY

Set this keyword to perform a grayscale, rather than binary, operation. Nonzero elements of the *Structure* parameter determine the shape of the structuring element. If the VALUES keyword is not present, all elements of the structuring element are 0.

PRESERVE_TYPE

Set this keyword to return the same type as the input array. The input array must be of type BYTE, UINT, or ULONG. This keyword only applies for grayscale erosion/dilation, and is mutually exclusive of the UINT and ULONG keywords.

UINT

Set this keyword to return an unsigned integer array. This keyword only applies for grayscale operations, and is mutually exclusive of the ULONG and PRESERVE_TYPE keywords.

ULONG

Set this keyword to return an unsigned longword integer array. This keyword only applies for grayscale operations and is mutually exclusive of the UINT and PRESERVE_TYPE keywords.

VALUES

An array of the same dimensions as *Structure* providing the values of the structuring element. The presence of this keyword implies a grayscale operation.

Example

The following code snippet reads a data file in the IDL Demo data directory containing an magnified image of grains of pollen. It then applies a threshold and a morphological opening operator with a 3 by 3 square kernel to the original image. Notice that much of the irregular borders of the grains have been smoothed by the opening operator.

```

;Read the image
READ_JPEG, DEMO_FILEPATH('pollens.jpg', $
    SUBDIR=['examples', 'demo', 'demodata']), a
;Apply the threshold
b = a ge 140b
;Load a simple color table
TEK_COLOR
TV, b, 0
;Apply opening operator
c = MORPH_OPEN(b, REPLICATE(1,3,3))
;Show the result
TV, c, 1
;Show pixels that have been removed in white
TV, c + b, 2

```

MORPH_THIN

The MORPH_THIN function performs a thinning operation on binary images. The thinning operator is implemented by first applying a hit or miss operator to the original image with a pair of structuring elements, and then subtracting the result from the original image.

In typical applications, this operator is repeatedly applied with the two structuring elements, while rotating them after each application, until the result remains unchanged.

Syntax

$$Result = \text{MORPH_THIN} (Image, HitStructure, MissStructure)$$

Arguments

Image

A one-, two-, or three-dimensional array upon which the thinning operation is to be performed. The image is treated as a binary image with all nonzero pixels considered as 1.

HitStructure

A one-, two-, or three-dimensional array to be used as the hit structuring element. The elements are interpreted as binary values — either zero or nonzero. This structuring element must have the same number of dimensions as the *Image* argument.

MissStructure

A one-, two-, or three-dimensional array to be used as the miss structuring element. The elements are interpreted as binary values — either zero or nonzero. This structuring element must have the same number of dimensions as the *Image* argument.

Note

It is assumed that the *HitStructure* and the *MissStructure* arguments are disjoint.

Keywords

None.

MORPH_TOPHAT

The MORPH_TOPHAT function applies the top-hat operator to a grayscale image. The top-hat operator is implemented by first applying the opening operator to the original image, then subtracting the result from the original image. Applying the top-hat operator provides a result that shows the bright peaks within the image.

Syntax

```
Result = MORPH_TOPHAT ( Image, Structure [, PRESERVE_TYPE=bytearray |  
/UINT | /ULONG] [, VALUES=array] )
```

Arguments

Image

A one-, two-, or three-dimensional array upon which the top-hat operation is to be performed.

Structure

A one-, two-, or three-dimensional array to be used as the structuring element. The elements are interpreted as binary values — either zero or nonzero. The structuring element must have the same number of dimensions as the *Image* argument.

Keywords

PRESERVE_TYPE

Set this keyword to return the same type as the input array. The input array must be of type BYTE, UINT, or ULONG. This keyword only applies for grayscale erosion/dilation, and is mutually exclusive of the UINT and ULONG keywords.

UINT

Set this keyword to return an unsigned integer array. This keyword is mutually exclusive of the ULONG and PRESERVE_TYPE keywords.

ULONG

Set this keyword to return an unsigned longword integer array. This keyword is mutually exclusive of the UINT and PRESERVE_TYPE keywords.

VALUES

An array of the same dimensions as the *Structure* argument providing the values of the structuring element. If the VALUES keyword is not present, all elements of the structuring element are 0.

Example

The following example illustrates an application of the top-hat operator to an image in the `examples/demo/demodata` directory:

```
;Read the image
READ_JPEG, DEMO_FILEPATH('pollens.jpg', $
    SUBDIR=['examples','demo','demodata']), a

;Show original
TVSCL, a, 0

;Radius of disc
r = 2

;Create a binary disc of given radius.
disc = SHIFT(DIST(2*r+1), r, r) LE r

;Apply top-hat operator
b = MORPH_TOPHAT(a, disc)

;Display stretched result.
tvsc1, b < 50, 1
```

MSG_CAT_CLOSE

The MSG_CAT_CLOSE procedure closes a catalog file from the stored cache.

Syntax

MSG_CAT_CLOSE, *object*

Arguments

object

The object reference returned from MSG_CAT_OPEN.

Keywords

None

MSG_CAT_COMPILE

The MSG_CAT_COMPILE procedure creates an IDL language catalog file.

Note

The locale is determined from the system locale in effect when compilation takes place.

Syntax

```
MSG_CAT_COMPILE, input[, output] [, LOCALE_ALIAS=string] [, /MBCS]
```

Arguments

input

The input file with which to create the catalog. The file is a text representation of the key/MBCS association. Each line in the file must have a key. The language string must then be surrounded by double quotes, then an optional comment.

For example:

```
VERSION "Version 1.0" My revision number of the file
```

There are 2 special tags, one of which must be included when creating the file.

```
APPLICATION (required)
```

```
SUB_QUERY (optional)
```

output

The optional output file name (including path if necessary) of the IDL language catalog file.

The naming convention for IDL language catalog files is as follows:

```
idl_ + "Application name" + _ + "Locale" + .cat
```

For example:

```
idl_envi_usa_eng.cat
```

If not set, a default filename is used based on the locale:

```
idl_[locale].cat
```


Keywords

LOCALE_ALIAS

Set this keyword to a scalar string containing any locale aliases for the locale on which the catalog is being compiled. A semi-colon is used to separate locales.

For example:

```
MSG_CAT_COMPILE, 'input.txt', 'idl_envi_usa_eng.cat', $  
LOCALE_ALIAS='C'
```

MBCS

If set, this procedure assumes language strings to be in MBCS format. The default is 8-bit ASCII.

MSG_CAT_OPEN

The MSG_CAT_OPEN function returns a catalog object for the given parameters if found. If a match is not found, an unset catalog object is returned. If unset, the [IDLffLanguageCat::Query](#) method will always return the empty string unless a default catalog is provided.

Syntax

```
Result = MSG_CAT_OPEN( application [, DEFAULT_FILENAME=filename]  
[, FILENAME=string] [, FOUND=variable] [, LOCALE=string] [, PATH=string]  
[, SUB_QUERY=value] )
```

Arguments

application

A scalar string representing the name of the desired application's catalog file.

Keywords

DEFAULT_FILENAME

Set this keyword to a scalar string containing the full path and filename of the catalog file to open if the initial request was not found.

FILENAME

Set this keyword to a scalar string containing the full path and filename of the catalog file to open. If this keyword is set, *application*, PATH and LOCALE are ignored.

FOUND

Set this keyword to a named variable that will contain 1 if a catalog file was found, 0 otherwise.

LOCALE

Set this keyword to the desired locale for the catalog file. If not set, the current locale is used.

PATH

Set this keyword to a scalar string containing the path to search for language catalog files. The default is the current directory.

SUB_QUERY

Set this keyword equal to the value of the SUB_QUERY key to search against. If a match is found, it is used to further sub-set the possible return catalog choices.

PARTICLE_TRACE

The `PARTICLE_TRACE` procedure traces the path of a massless particle through a vector field. The function allows the user to specify a set of starting points and a vector field. The input seed points can come from any vertex-producing process. The points are tracked by treating the vector field as a velocity field and integrating. Each path is tracked until the path leaves the input volume or a maximum number of steps is reached. The vertices generated along the paths are returned packed into a single array along with a polyline connectivity array. The polyline connectivity array organizes the vertices into separate paths (one per seed). Each path has an orientation. The initial orientation may be set using the `SEED_NORMAL` keyword. As a path is tracked, the change in the normal is also computed and may be returned to the user as an optional argument. Path output can be passed directly to an `IDLgrPolyline` object or passed to the `STREAMLINE` procedure for generation of orientated ribbons. Control over aspects of the integration (e.g. method or stepsize) is also provided.

Syntax

```
PARTICLE_TRACE, Data, Seeds, Verts, Conn [, Normals]
[, MAX_ITERATIONS=value] [, ANISOTROPY=array]
[, INTEGRATION={0 | 1}] [, SEED_NORMAL=vector] [, TOLERANCE=value]
[, MAX_STEPSIZE=value] [, /UNIFORM]
```

Arguments

Data

Input data array. This array can be of dimensions $[2, dx, dy]$ for two-dimensional vector fields or $[3, dx, dy, dz]$ for three-dimensional vector fields.

Seeds

Input array of seed points ($[3, n]$ or $[2, n]$).

Verts

Array of output path vertices ($[3, n]$ or $[2, n]$ array of floats).

Conn

Output path connectivity array in `IDLgrPolyline POLYLINES` keyword format. There is one set of line segments in this array for each input seed point.

Normals

Output normal estimate at each output vertex ($[3, n]$ array of floats).

Keywords

ANISOTROPY

Set this input keyword to a two- or three- element array describing the distance between grid points in each dimension. The default value is $[1.0, 1.0, 1.0]$ for three-dimensional data and $[1.0, 1.0]$ for two-dimensional data.

INTEGRATION

Set this keyword to one of the following values to select the integration method:

- 0 = 2nd order Runge-Kutta
- 1 = 4th order Runge-Kutta

The default is zero.

SEED_NORMAL

Set this keyword to a three-element vector which selects the initial normal for the paths. The default value is $[0.0, 0.0, 1.0]$. This keyword is ignored for two-dimensional data.

TOLERANCE

This keyword is used with adaptive step-size control in the 4th order Runge-Kutta integration scheme. It is ignored if the UNIFORM keyword is set or the 2nd order Runge-Kutta scheme is selected.

MAX_ITERATIONS

This keyword specifies the maximum number of line segments to return for each path. The default value is 200.

MAX_STEPSIZE

This keyword specifies the maximum path step size. The default value is 1.0.

UNIFORM

If this keyword is set, the step size will be set to a fixed value, set via the MAX_STEPSIZE keyword. If this keyword is not specified, and TOLERANCE is either unspecified or inapplicable, then the step size is computed based on the velocity at the current point on the path according to the formula:

$\text{stepsize} = \text{MIN}(\text{MaxStepSize}, \text{MaxStepSize}/\text{MAX}(\text{ABS}(U), \text{ABS}(V), \text{ABS}(W)))$
where (U,V,W) is the local velocity vector.

QUERY_IMAGE

The QUERY_IMAGE function reads the header of a file and determines if it is recognized as an image file. If it is an image file, an optional structure containing information about the image is returned.

Syntax

```
Result = QUERY_IMAGE ( Filename[, Info] [, CHANNELS=variable]
[, DIMENSIONS=variable] [, HAS_PALETTE=variable]
[, IMAGE_INDEX=index] [, NUM_IMAGES=variable] [, PIXEL_TYPE=variable]
[, SUPPORTED_READ=variable] [, SUPPORTED_WRITE=variable]
[, TYPE=variable] )
```

Return Value

Result is a long with the value of 1 if the query was successful (the file was recognized as an image file) or 0 on failure. The return status will indicate failure for files that contain formats that are not supported by the corresponding READ_ routine, even though the file may be valid outside the IDL environment.

Arguments

Filename

A scalar string containing the name of the file to query.

Info

An optional anonymous structure containing information about the image. This structure is valid only when the return value of the function is 1. The Info structure for all image types has the following fields:

Tag	Type
CHANNELS	Long
DIMENSIONS	Two-dimensional long array
FILENAME	Scalar string
HAS_PALETTE	Integer

Table 2: The Info Structure for All Image Types

Tag	Type
IMAGE_INDEX	Long
NUM_IMAGES	Long
PIXEL_TYPE	Integer
TYPE	Scalar string

Table 2: The Info Structure for All Image Types

Keywords

CHANNELS

Set this keyword to a named variable to retrieve the number of channels in the image.

DIMENSIONS

Set this keyword to a named variable to retrieve the image dimensions as a two-dimensional array.

HAS_PALETTE

Set this keyword to a named variable to equal to 1 if a palette is present, else 0.

IMAGE_INDEX

Set this keyword to the index of the image to query from the file. The default is 0, the first image.

NUM_IMAGES

Set this keyword to a named variable to retrieve the number of images in the file.

PIXEL_TYPE

Set this keyword to a named variable to retrieve the IDL Type Code of the image pixel format. See the documentation for the `SIZE` routine for a complete list of IDL Type Codes.

The valid types for `PIXEL_TYPE` are:

- 1 = Byte
- 2 = Integer
- 3 = Longword Integer

- 4 = Floating Point
- 5 = Double-precision Floating Point
- 12 = Unsigned Integer
- 13 - Unsigned Longword Integer
- 14 - 64-bit Integer
- 15 - Unsigned 64-bit Integer

SUPPORTED_READ

Set this keyword to a named variable to retrieve a string array of image types recognized by `READ_IMAGE`. If the `SUPPORTED_READ` keyword is used the filename and info arguments are optional.

SUPPORTED_WRITE

Set this keyword to a named variable to retrieve a string array of image types recognized by `WRITE_IMAGE`. If the `SUPPORTED_WRITE` keyword is used the filename and info arguments are optional.

TYPE

Set this keyword to a named variable to retrieve the image type as a scalar string. Valid return values are:

- BMP
- GIF
- JPEG
- PNG
- PPM
- SRF
- TIFF
- DICOM

QUERY_WAV

The `QUERY_WAV` function checks that the file is actually a .WAV file and that the `READ_WAV` function can read the data in the file. Optionally, it can return additional information about the data in the file. This function returns the value of 1 if the query was successful (and the file type was correct) or 0 on failure.

Syntax

Result = `QUERY_WAV` (*Filename* , *Info*)

Arguments

Filename

A scalar string containing the full pathname of the .WAV file to read.

Info

An anonymous structure containing information about the data in the file. The fields are defined as:

Tag	Type	Definition
CHANNELS	INT	Number of data channels in the file.
SAMPLES_PER_SEC	LONG	Data sampling rate in samples per second.
BITS_PER_SAMPLE	INT	Number of valid bits in the data.

Table 5-6: The Info Structure for Info Fields

Keywords

None.

READ_BINARY

The READ_BINARY function reads the contents of a binary file using a passed template or basic command line keywords. The result is an array or anonymous structure containing all of the entities read from the file. Data is read from the given filename or from the current file position in the open file pointed to by FileUnit. If no template is provided, keywords can be used to read a single IDL array of data.

Note

The READ_BINARY function does not work on VMS platforms due to limitations in the POINT_LUN procedure. For more information, see [POINT_LUN](#).

Syntax

```
Result = READ_BINARY ([Filename] | FileUnit [, TEMPLATE=template] |  
[[, DATA_START=value] [, DATA_TYPE=typecodes] [, DATA_DIMS=array]  
[, ENDIAN=string] )
```

Arguments

Filename

A scalar string containing the name of the binary file to read. If *filename* and file unit are not specified, a dialog allows the user to choose a file.

FileUnit

A scalar containing an open IDL file unit number to read from.

Keywords

DATA_DIMS

Set this keyword to a scalar or array of up to eight elements specifying the size of the data to be read and returned. For example, DATA_DIMS=[512,512] specifies that a two-dimensional, 512 by 512 array be read and returned. DATA_DIMS=0 specifies that a single, scalar value be read and returned. Default is -1, which, if a TEMPLATE is not supplied that specifies otherwise, indicates that READ_BINARY will read to end-of-file and store the result in a 1D array.

DATA_START

Set this keyword to specify where to begin reading in a file. This value is as an offset, in bytes, that will be applied to the initial position in the file. Default is 0.

DATA_TYPE

Set this keyword to an IDL typecode of the data to be read. See documentation for the IDL SIZE function for a listing of typecodes. Default is 1 (IDL's BYTE typecode).

ENDIAN

Set this keyword to one of three string values: "big", "little" or "native" which specifies the byte ordering of the file to be read. If the computer running READ_BINARY uses byte ordering that is different than that of the file, READ_BINARY will swap the order of bytes in multi-byte data types read from the file. (Default: "native" = perform no byte swapping.)

TEMPLATE

Set this keyword to a template structure describing the file to be read. A template can be created using BINARY_TEMPLATE. The TEMPLATE keyword cannot be used simultaneously with keywords DATA_START, DATA_TYPE, DATA_DIMS, or ENDIAN.

When a template is used with READ_BINARY, the result of a successful call to READ_BINARY is a structure containing fields specified by the template.

If a template is not used with READ_BINARY, the result of a successful call to READ_BINARY is an array.

READ_IMAGE

The READ_IMAGE function reads the image contents of a file and returns the image in an IDL variable. If the image contains a palette it can be returned as well in three IDL variables. READ_IMAGE returns the image in the form of a two-dimensional array (for grayscale images) or a (3, n, m) array (for TrueColor images). READ_IMAGE can read most types of image files supported by IDL. See QUERY_IMAGE for a list of supported formats.

Syntax

```
Result = READ_IMAGE (Filename [, Red, Green, Blue]  
[, ALLOWED_FORMATS=string] [, FORMAT=string] [, IMAGE_INDEX=index])
```

Return Value

Result is the image array read from the file or scalar value of -1 if the file could not be read.

Arguments

Filename

A scalar string containing the name of the file to read.

Red

An optional named variable to receive the red channel of the color table if a color table exists.

Green

An optional named variable to receive the green channel of the color table if a color table exists.

Blue

An optional named variable to receive the blue channel of the color table if a color table exists.

Keywords

ALLOWED_FORMATS

Set this keyword to a scalar or array of format types `READ_IMAGE` will be allowed to read. The default is all known image types.

FORMAT

Set this keyword to a scalar string of the image type to read. This will force `READ_IMAGE` to attempt to read the file as the given format type.

IMAGE_INDEX

Set this keyword to the index of the image to read from the file. The default is 0, the first image.

READ_WAV

The READ_WAV function reads the audio stream from the named .WAV file. Optionally, it can return the sampling rate of the audio stream.

Syntax

Result = READ_WAV (*Filename* [, *Rate*])

Return Value

In the case of a single channel stream, the returned variable is a BYTE or INT (depending on the number of bits per sample) one-dimensional array. In the case of a file with multiple channels, a similar two-dimensional array is returned, with the leading dimension being the channel number.

Arguments

Filename

A scalar string containing the full pathname of the .WAV file to read.

Rate

Returns an IDL long containing the sampling rate of the stream in samples per second.

Keywords

None.

STRCMP

The STRCMP function performs string comparisons between its two String arguments, returning True (1) for those that match and False (0) for those that do not. Normally, the IDL equality operator (EQ) is used for such comparisons, but STRCMP can optionally perform case-insensitive comparisons and can be limited to compare only the first N characters of the two strings, both of which require extra steps using the EQ operator.

Syntax

```
Result = STRCMP (String1, String2 [, N] [, /FOLD_CASE] )
```

Return Value

If all of the arguments are scalar, the result is scalar. If one of the arguments is an array, the result is an integer with the same structure. If more than one argument is an array, the result has the structure of the smallest array. Each element of the result contains True (1) if the corresponding elements of String1 and String2 are the same, and False (0) otherwise.

Arguments

String1, String2

The strings to be compared.

N

Normally String1 and String2 are compared in their entirety. If N is specified, the comparison is made on at most the first N characters of each string.

Keywords

FOLD_CASE

String comparison is normally a case-sensitive operation. Set FOLD_CASE to perform case-insensitive comparisons instead.

Example

Compare two strings in a case-insensitive manner, considering only the first 3 characters:


```
Result = STRCMP('Moose', 'moo', 3, /FOLD_CASE)  
PRINT, Result
```

IDL Output

1

STREAMLINE

The STREAMLINE procedure generates the visualization graphics from a path. The output is a polygonal ribbon which is tangent to a vector field along its length. The ribbon is generated by placing a line at each vertex in the direction specified by each normal value multiplied by the anisotropy factor. The input normal array is not normalized before use, making it possible to vary the ribbon width as well.

Syntax

```
STREAMLINE, Verts, Conn, Normals, Outverts, Outconn [, ANISOTROPY=array]  
[, SIZE=vector] [, PROFILE=array]
```

Arguments

Verts

Input array of path vertices ([3, *n*] array).

Conn

Input path connectivity array in IDLgrPolyline POLYLINES keyword format. There is one set of line segments in this array for each streamline.

Normals

Normal estimate at each input vertex ([3, *n*] array).

Outverts

Output vertices ([3xM] float array). Useful if the routine is to be used with Direct Graphics or the user wants to manipulate the data directly.

Outconn

Output polygonal connectivity array to match the output vertices.

Keywords

ANISOTROPY

Set this input keyword to a three-element array describing the distance between grid points in each dimension. The default value is [1.0, 1.0, 1.0]

SIZE

Set this keyword to a vector of values (one for each path point). These values are used to specify the width of the ribbon or the size of profile at each point along its path. This keyword is generally used to convey additional data parameters along the streamline.

PROFILE

Set this keyword an array of two-dimensional points which are treated as the cross section of the ribbon instead of a line segment. If the first and last points in the array are the same, a closed profile is generated. The profile is placed at each path vertex in the plane perpendicular to the line connecting each path vertex with the vertex normal defining the up direction. This allows for the generation of streamtubes and other geometries.

STREGEX

The STREGEX procedure performs regular expression matching against the strings contained in *StringExpression*. STREGEX can perform either a simple boolean True/False evaluation of whether a match occurred, or it can return the position and offset within the strings for each match. The regular expressions accepted by this routine, which correspond to “Posix Extended Regular Expressions”, are similar to those used by such UNIX tools as *egrep*, *lex*, *awk*, and *Perl*.

For more information about regular expressions, see “[Learning About Regular Expressions](#)” in Chapter 9 of *Building IDL Applications*.

STREGEX is based on the *regex* package written by Henry Spencer, modified by RSI only to the extent required to integrate it into IDL. This package is freely available at <ftp://zoo.toronto.edu/pub/regex.shar>.

Syntax

```
Result = STREGEX (StringExpression, RegularExpression [, /BOOLEAN |  
, /EXTRACT | , LENGTH=variable [, /SUBEXPR]] [, /FOLD_CASE] )
```

Return Value

By default, STREGEX returns the position and length of the matched string within *StringExpression*. If no match is found, -1 is returned for both of these. Optionally, it can return a boolean True/False result of the match, or the matched strings.

Arguments

StringExpression

String to be matched.

RegularExpression

A scalar string containing the regular expression to match. See “[Learning About Regular Expressions](#)” in Chapter 9 of *Building IDL Applications* for a description of the meta characters that can be used in a regular expression.

Keywords

BOOLEAN

Normally, STREGEX returns the position of the first character in StringExpression that matches RegularExpression. Setting BOOLEAN modifies this behavior to simply return a True/False value indicating if a match occurred or not.

EXTRACT

Normally, STREGEX returns the position of the first character in StringExpression that matches RegularExpression. Setting EXTRACT modifies this behavior to simply return the matched substrings. The EXTRACT keyword cannot be used with either BOOLEAN or LENGTH.

FOLD_CASE

Regular expression matching is normally a case-sensitive operation. Set FOLD_CASE to perform case-insensitive matching instead.

LENGTH

If present, specifies a variable to receive the lengths of the matches. Together with this result of this function, which contains the starting points of the matches in StringExpression, LENGTH can be used with the STRMID function to extract the matched substrings. The LENGTH keyword cannot be used with either BOOLEAN or EXTRACT.

SUBEXPR

By default, STREGEX only reports the overall match. Setting SUBEXPR causes it to report the overall match as well as any subexpression matches. A subexpression is any part of a regular expression written within parentheses. For example, the regular expression '(a)(b)(c+)' has 3 subexpressions, whereas the functionally equivalent 'abc+' has none. The SUBEXPR keyword cannot be used with BOOLEAN.

If a subexpression participated in the match several times, the reported substring is the last one it matched. Note, as an example in particular, that when the regular expression '(b*)+' matches 'bbb', the parenthesized subexpression matches the three 'b's and then an infinite number of empty strings following the last 'b', so the reported substring is one of the empties. This occurs because the '*' matches *zero or more* instances of the character that precedes it.

In order to return multiple positions and lengths for each input, the result from SUBEXPR has a new first dimension added compared to StringExpression.

Examples

Example 1

To match a string starting with an “a”, followed by a “b”, followed by 1 or more “c”:

```
pos = STREGEX('aaabccc', 'abc+', length=len)
PRINT, STRMID('aaabccc', pos, len)
```

IDL Prints:

```
abccc
```

To perform the same match, and also find the locations of the three parts:

```
pos = STREGEX('aaabccc', '(a)(b)(c+)', length=len, /SUBEXPR)
print, STRMID('aaabccc', pos, len)
```

IDL Prints:

```
abccc a b ccc
```

Or more simply:

```
print, STREGEX('aaabccc', '(a)(b)(c+)', /SUBEXPR, /EXTRACT)
```

IDL Prints:

```
abccc a b ccc
```

Example 2

This example searches a string array for words of any length beginning with “f” and ending with “t” without the letter “o” in between:

```
str = ['foot', 'Feet', 'fate', 'FAST', 'ferret', 'affluent']
PRINT, STREGEX(str, '^f[^o]*t$', /EXTRACT, /FOLD_CASE)
```

This statement results in:

```
Feet FAST ferret
```

Note the following about this example:

- Unlike the * wildcard character used by STRMATCH, the * meta character used by STREGEX applies to the item directly on its left, which in this case is [^o], meaning “any character except the letter ‘o’”. Therefore, [^o]* means “zero or more characters that are not ‘o’”, whereas the following statement would find only words whose second character is not “o”:

```
PRINT, str[WHERE(STRMATCH(str, 'f[!o]*t', /FOLD_CASE) EQ 1)]
```

- The anchors (^ and \$) tell STREGEX to find only words that begin with “f” and end with “t”. If we left out the ^ anchor in the above example, STREGEX would also return “ffluent” (a substring of “affluent”). Similarly, if we left out the \$ anchor, STREGEX would also return “fat” (a substring of “fate”).

STRJOIN

The STRJOIN function collapses a string scalar or array into merged strings. This function reduces the rank of its input array by one dimension. The strings in the removed first dimension are concatenated into a single string using the string in *Delimiter* to separate them.

Syntax

Result = STRJOIN (*String* [, *Delimiter*] [, /SINGLE])

Arguments

String

A string scalar or array to be collapsed into merged strings.

Delimiter

The separator string to use between the joined strings. If *Delimiter* is not specified, an empty string is used.

Keywords

SINGLE

If SINGLE is set, the entire String is joined into a single scalar string result.

Example

Replace all the blanks in a sentence with colons:

```
str = 'Out, damned spot! Out I say!'
print, (STRJOIN(STRSPLIT(str, /EXTRACT), ':'))
```

IDL Output

```
Out,:damned:spot!:Out:I:say!
```


STRMATCH

The STRMATCH function compares its search string, which can contain wildcard characters, against the input string expression. The result is an array with the same structure as the input string expression. Those elements that match the corresponding input string are set to True (1), and those that do not match are set to False (0).

The wildcards understood by STRMATCH are similar to those used by the standard UNIX shell:

Wildcard Character	Description
*	Matches any string, including the null string.
?	Matches any single character.
[...]	Matches any one of the enclosed characters. A pair of characters separated by “-” matches any character lexically between the pair, inclusive. If the first character following the opening [is a !, any character not enclosed is matched. To prevent one of these characters from acting as a wildcard, it can be quoted by preceding it with a backslash character (e.g. “*” matches the asterisk character). Quoting any other character (including \ itself) is equivalent to the character (e.g. “\a” is the same as “a”).

Table 5-7: Wildcard Characters used by STRMATCH

Syntax

Result = STRMATCH(*String*, *SearchString* [, /FOLD_CASE])

Arguments

String

The String to be matched.

SearchString

The search string, which can contain wildcard characters as discussed above.

Keywords

FOLD_CASE

The comparison is usually case sensitive. Setting the FOLD_CASE keyword causes a case insensitive match to be done instead.

Examples

Example 1

Find all 4-letter words in a string array that begin with “f” or “F” and end with “t” or “T”:

```
str = ['foot', 'Feet', 'fate', 'FAST', 'ferret', 'fort']
PRINT, str[WHERE(STRMATCH(str, 'f??t', /FOLD_CASE) EQ 1)]
```

This results in:

```
foot Feet FAST fort
```

Example 2

Find words of any length that begin with “f” and end with “t”:

```
str = ['foot', 'Feet', 'fate', 'FAST', 'ferret', 'fort']
PRINT, str[WHERE(STRMATCH(str, 'f*t', /FOLD_CASE) EQ 1)]
```

This results in:

```
foot Feet FAST ferret fort
```

Example 3

Find 4-letter words beginning with “f” and ending with “t”, with any combination of “o” and “e” in between:

```
str = ['foot', 'Feet', 'fate', 'FAST', 'ferret', 'fort']
PRINT, str[WHERE(STRMATCH(str, 'f[eo][eo]t', /FOLD_CASE) EQ 1)]
```

This results in:

```
foot Feet
```

Example 4

Find all words beginning with “f” and ending with “t” whose second character is not the letter “o”:

```
str = ['foot', 'Feet', 'fate', 'FAST', 'ferret', 'fort']
PRINT, str[WHERE(STRMATCH(str, 'f[!o]*t', /FOLD_CASE) EQ 1)]
```

This results in:

```
Feet FAST ferret
```

STRSPLIT

The STRSPLIT function splits its input *String* argument into separate substrings, according to the specified delimiter or regular expression. By default, the position of the substrings is returned. The EXTRACT keyword can be used to cause STRSPLIT to return an array containing the substrings.

Syntax

```
Result = STRSPLIT ( String [, Pattern] [, ESCAPE=string | , /REGEX  
| , /FOLD_CASE] [, /EXTRACT | , LENGTH=variable] [, /PRESERVE_NULL] )
```

Arguments

String

A scalar string to be split into substrings.

Pattern

Pattern can contain one of two types of information:

- A string containing the character codes that are considered to be separators. In this case, IDL performs a simple string search for those characters. This method is simple and fast.
- A regular expression, as implemented by the STREGEX function, which is used by IDL to match the separators. This method is slower and more complex, but can handle extremely complicated input strings.

Pattern is an optional argument. If it is not specified, STRSPLIT defaults to splitting on spans of whitespace (space or tab characters) in *String*.

Keywords

ESCAPE

When doing simple pattern matching, the ESCAPE keyword can be used to specify any characters that should be considered to be “escape” characters. Preceding any character with an escape character prevents STRSPLIT from treating it as a separator character even if it is found in *Pattern*.

Note that if the EXTRACT keyword is set, STRSPLIT will automatically remove the escape characters from the resulting substrings. If EXTRACT is not specified,

STRSPLIT cannot perform this editing, and the returned position and offsets will include the escape characters.

For example:

```
print, STRSPLIT('a\b', ' ', ' ', ESCAPE='\', /EXTRACT)
```

IDL prints:

```
a,b
```

ESCAPE cannot be specified with the FOLD_CASE or REGEX keywords.

EXTRACT

By default, STRTRIM returns an array of character offsets into *String* that indicate where the substrings are located. These offsets, along with the lengths available from the LENGTH keyword can be used later with STRMID to extract the substrings. Set EXTRACT to bypass this step, and cause STRSPLIT to return the substrings. EXTRACT cannot be specified with the LENGTH keyword.

FOLD_CASE

Indicates that the regular expression matching should be done in a case-insensitive fashion. FOLD_CASE can only be specified if the REGEX keyword is set, and cannot be used with the ESCAPE keyword.

LENGTH

Set this keyword to a named variable to receive the lengths of the substrings. Together with this result of this function, LENGTH can be used with the STRMID function to extract the matched substrings. The LENGTH keyword cannot be used with the EXTRACT keyword.

PRESERVE_NULL

Normally, STRSPLIT will not return null length substrings unless there are no non-null values to report, in which case STRSPLIT will return a single null string. Set PRESERVE_NULL to cause all null substrings to be returned.

REGEX

For complex splitting tasks, the REGEX keyword can be specified. In this case, *Pattern* is taken to be a regular expression to be matched against *String* to locate the separators. If REGEX is specified and *Pattern* is not, the default *Pattern* is the regular expression:

```
'[ ' + STRING(9B) + ']+'
```

which means “any series of one or more space or tab characters” (9B is the byte value of the ASCII TAB character).

Note that the default *Pattern* contains a space after the [character.

The REGEX keyword cannot be used with the ESCAPE keyword.

Examples

Example 1

To split a string on spans of whitespace and replace them with hyphens:

```
Str = 'STRSPLIT chops up strings.'
print, STRJOIN(STRSPLIT(Str, /EXTRACT), '-')
```

IDL Output

```
STRSPLIT-chops-up-strings.
```

Example 2

As an example of a more complex splitting task that can be handled with the simple character-matching mode of STRSPLIT, consider a sentence describing different colored ampersand characters. For unknown reasons, the author used commas to separate all the words, and used ampersands or backslashes to escape the commas that actually appear in the sentence (which therefore should not be treated as separators). The unprocessed string looks like:

```
Str = 'There,was,a,red,&&&,a,yellow,&&\, ,and,a,blue,\&.'
```

We use STRSPLIT to break this line apart, and STRJOIN to reassemble it as a standard blank-separated sentence:

```
S = STRSPLIT(Str, ',', ESCAPE='&\', /EXTRACT)
PRINT, STRJOIN(S, ' ')
```

IDL Output

```
There was a red &, a yellow &, and a blue &.
```

Example 3

Finally, suppose you had a complicated string, in which every token was preceded by the count of characters in that token, with the count enclosed in angle brackets:

```
str = '<4>What<1>a<7>tangled<3>web<2>we<6>weave.'
```

This is too complex to handle with simple character matching, but can be easily handled using the regular expression '<[0-9]+>' to match the separators. This regular expression can be read as “an opening angle bracket, followed by one or more numeric characters between 0 and 9, followed by a closing angle bracket.” The STRJOIN function is used to glue the resulting substrings back together:

```
S = STRSPLIT(str, '<[0-9]+>', /EXTRACT, /REGEX)
PRINT, STRJOIN(S, ' ')
```

IDL Output

```
What a tangled web we weave.
```

STRUCT_HIDE

The IDL `HELP` procedure displays information on all known structures or object classes when used with the `STRUCTURES` or `OBJECTS` keywords. Although this is usually the desired behavior, authors of large vertical applications or library routines may wish to prevent IDL from displaying information on structures or objects that are not part of their public interface, but which exist solely in support of the internal implementation. The `STRUCT_HIDE` procedure is used to mark such structures or objects as “hidden”. Items so marked are not displayed by `HELP` unless the user sets the `FULL` keyword, but are otherwise unaltered.

Note

`STRUCT_HIDE` is primarily intended for use with named structures or objects. Although it can be safely used with anonymous structures, there is no visible benefit to doing so as anonymous structures are hidden by default.

Tip

Authors of objects will often place a call to `STRUCT_HIDE` in the `__DEFINE` procedure that defines the structure.

Syntax

```
STRUCT_HIDE, Arg1 [, Arg2, ..., Argn]
```

Arguments

Arg₁, ..., Arg_n

If an argument is a variable of one of the following types, its underlying structure and/or object definition is marked as being hidden from the `HELP` procedure’s default output:

- Structure
- Pointer that refers to a heap variable of structure type
- Object Reference

Any arguments that are not one of these types are quietly ignored. No change is made to the value of any argument.

Keywords

None.

Example

To create a named structure called “bullwinkle” and prevent it from appearing in the HELP procedure’s default output:

```
tmp = { bullwinkle, moose:1, squirrel:0 }  
STRUCT_HIDE, tmp
```

TETRA_CLIP

The TETRA_CLIP function clips a tetrahedral mesh to an arbitrary plane in space and returns a tetrahedral mesh of the remaining portion. An auxiliary array of data may also be passed and clipped. This array can have multiple values for each vertex (the trailing array dimension must match the number of vertices in the Vertsin array).

A tetrahedral connectivity array consists of groups of four vertex index values. Each set of four index values specifies four vertices which define a single tetrahedron.

Syntax

```
Result = TETRA_CLIP ( Plane, Vertsin, Connin, Vertsout, Connout  
[, AUXDATA_IN=array, AUXDATA_OUT=variable] [, CUT_VERTS=variable] )
```

Return Value

The return value is the number of tetrahedra returned.

Arguments

Plane

Input four-element array describing the equation of the plane to be clipped to. The elements are the coefficients (a, b, c, d) of the equation $ax+by+cz+d=0$.

Vertsin

Input array of tetrahedral vertices [3, n].

Connin

Input tetrahedral mesh connectivity array.

Vertsout

Output array of tetrahedral vertices [3, n].

Connout

Output tetrahedral mesh connectivity array.

Keywords

AUXDATA_IN

Input array of auxiliary data. If present, these values are interpolated and returned through AUXDATA_OUT. The trailing array dimension must match the number of vertices in the Vertsin array.

AUXDATA_OUT

Set this keyword to a named variable to contain an output array of interpolated auxiliary data.

CUT_VERTS

Set this keyword to a named variable to contain an output array of vertex indices (into Vertsout) of the vertices which are considered to be 'on' the clipped surface.

TETRA_SURFACE

The TETRA_SURFACE function extracts a polygonal mesh as the exterior surface of a tetrahedral mesh. The output of this function is a polygonal mesh connectivity array that can be used with the input Verts array to display the outer surface of the tetrahedral mesh.

Syntax

Result = TETRA_SURFACE (*Verts*, *Connin*)

Return Value

Returns a polygonal mesh connectivity array. When used with the input vertex array, this function yields the exposed tetrahedral mesh surface.

Arguments

Verts

Array of vertices [3, *n*].

Connin

Tetrahedral connectivity array.

TETRA_VOLUME

The TETRA_VOLUME function computes properties of a tetrahedral mesh array. The basic property is the volume. An auxiliary data array may be supplied which specifies weights at each vertex which are interpolated through the volume during integration. Higher order moments (with respect to the X, Y, and Z axis) may be computed as well (with or without weights).

Syntax

```
Result = TETRA_VOLUME ( Verts, Conn [, AUXDATA=array]  
[, MOMENT=variable] )
```

Return Value

Returns the cumulative (weighted) volume of the tetrahedrons in the mesh.

Arguments

Verts

Array of vertices [3, *n*].

Conn

Tetrahedral connectivity array.

Keywords

AUXDATA

Array of input auxiliary data (one value per vertex). If present, these values are used to weight a vertex. The volume area integral will linearly interpolate these values. The volume integral will linearly interpolate these values within each tetrahedra. The default weight is 1.0 which results in a basic volume.

MOMENT

Set this keyword to a named variable that will contain a three-element float vector which corresponds to the first order moments computed with respect to the X, Y and Z axis. The computation is:

$$\vec{m} = \sum_{ntetras} v_i \vec{c}_i$$

where v is the (weighted) volume of the tetrahedron and c is the centroid of the tetrahedron, thus

$$\vec{m}/volume$$

yields the (weighted) centroid of the tetrahedral mesh.

VALUE_LOCATE

The VALUE_LOCATE function finds the intervals within a given monotonic vector that brackets a given set of one or more search values. This function is useful for interpolation and table-lookup, and is an adaptation of the locate() routine in Numerical Recipes. VALUE_LOCATE uses the bisection method to locate the interval.

Syntax

Result = VALUE_LOCATE (*Vector*, *Value*)

Return Value

Each return value, Result [*i*], is an index, *j*, into Vector, corresponding to the interval into which the given Value [*i*] falls. The returned values are in the range $-1 \leq j \leq N-1$, where *N* is the number of elements in the input vector.

If Vector is monotonically increasing, the result *j* is:

if $j = -1$ Value [*i*] < Vector [0]
 if $0 \leq j < N-1$ Vector [*j*] ≤ Value [*i*] < Vector [*j*+1]
 if $j = N-1$ Vector [*N*-1] ≤ Value [*i*]

If Vector is monotonically decreasing

if $j = -1$ Vector [0] ≤ Value [*i*]
 if $0 \leq j < N-1$ Vector [*j*+1] ≤ Value [*i*] < Vector [*j*]
 if $j = N-1$ Value [*i*] < Vector [*N*-1]

Arguments

Vector

A vector of monotonically increasing or decreasing values. Vector may be of type string, or any numeric type except complex, and may not contain the value NaN (not-a-number).

Value

The value for which the location of the intervals is to be computed. Value may be either a scalar or an array. The return value will contain the same number of elements as this parameter.

Keywords

None.

Example

```
; Define a vector of values.  
vec = [2,5,8,10]  
  
; Compute location of other values within that vector.  
loc = VALUE_LOCATE(vec, [0,3,5,6,12])  
PRINT, loc
```

IDL prints:

```
-1  0  1  1  3
```


VECTOR_FIELD

The VECTOR_FIELD procedure is used to place colored, oriented vectors of specified length at each vertex in an input vertex array. The output can be sent directly to an IDLgrPolyline object. The generated display is generally referred to as a hedgehog display and is used to convey various aspects of a vector field.

Syntax

```
VECTOR_FIELD, Field, Outverts, Outconn [, ANISOTROPY=array]  
[, SCALE=value] [, VERTICES=array]
```

Arguments

Field

Input vector field array. This can be a $[3, x, y, z]$ array or a $[2, x, y]$ array. The leading dimension is the vector quantity to be displayed.

Outverts

Output vertex array ($[3, N]$ or $[2, N]$ array of floats). Useful if the routine is to be used with Direct Graphics or the user wants to manipulate the data directly.

Outconn

Output polyline connectivity array to be applied to the output vertices.

Keywords

ANISOTROPY

Set this keyword to a two- or three-element array describing the distance between grid points in each dimension. The default value is $[1.0, 1.0, 1.0]$ for three-dimensional data and $[1.0, 1.0]$ for two-dimensional data.

SCALE

Set this keyword to a scalar scaling factor. All vector lengths are multiplied by this value. The default is 1.0.

VERTICES

Set this keyword to a $[3, n]$ or $[2, n]$ array of points. If this keyword is set, the vector field is interpolated at these points. The resulting interpolated vectors are displayed as

line segments at these locations. If the keyword is not set, each spatial sample point in the input Field grid is used as the base point for a line segment.

WATERSHED

The WATERSHED function applies the morphological watershed operator to a grayscale image. This operator segments images into watershed regions and their boundaries. Considering the gray scale image as a surface, each local minimum can be thought of as the point to which water falling on the surrounding region drains. The boundaries of the watersheds lie on the tops of the ridges. This operator labels each watershed region with a unique index, and sets the boundaries to zero.

Typically, morphological gradients, or images containing extracted edges are used for input to the watershed operator. Noise and small unimportant fluctuations in the original image can produce spurious minima in the gradients, which leads to oversegmentation. Smoothing, or manually marking the seed points are two approaches to overcoming this problem. For further reading, see Dougherty, “An Introduction to Morphological Image Processing”, SPIE Optical Engineering Press, 1992

Syntax

Result = WATERSHED (*Image* [, CONNECTIVITY={4 | 8}])

Return Value

Returns an image of the same dimensions as the input image. Each pixel of the result will be either zero if the pixel falls along the segmentation between basins, or the identifier of the basin in which that pixel falls.

Arguments

Image

The two-dimensional image to be segmented. *Image* is converted to byte type if necessary.

Keywords

CONNECTIVITY

Set this keyword to either 4 (to select 4-neighbor connectivity) or 8 (to select 8-neighbor connectivity). Connectivity indicates which pixels in the neighborhood of a given pixel are sampled during the segmentation process. 4-neighbor connectivity samples only the pixels that are immediately adjacent horizontally and vertically. 8-

neighbor connectivity samples the diagonally adjacent neighbors in addition to the immediate horizontal and vertical neighbors. The default is 4-neighbor connectivity.

Example

The following code snippet crudely segments the grains in the data file in the IDL Demo data directory containing an magnified image of grains of pollen.

It inverts the image, because the watershed operator finds holes, and the grains of pollen are bright. Next, the morphological closing operator is applied with a disc of radius 9, contained within a 19 by 19 kernel, to eliminate holes in the image smaller than the disc. The watershed operator is then applied to segment this image. The borders of the watershed images, which have pixel values of zero, are then merged with the original image and displayed as white.

```

;Radius of disc...
r = 9

;Create a disc of radius r
disc = SHIFT(DIST(2*r+1), r, r) LE r

;Read the image
READ_JPEG, DEMO_FILEPATH('pollens.jpg', $
    SUBDIR=['examples', 'demo', 'demodata']), a

;Invert the image
b = MAX(a) - a

TVSCL, b, 0

;Remove holes of radii less than r
c = MORPH_CLOSE(b, disc, /GRAY)

TVSCL, c, 1

;Create watershed image
d = WATERSHED(c)

;Display it, showing the watershed regions
TVSCL, d, 2

;Merge original image with boundaries of watershed regions
e = a > (MAX(a) * (d EQ 0b))

TVSCL, e, 3

```

WRITE_IMAGE

The WRITE_IMAGE procedure writes an image and its color table vectors, if any, to a file of a specified type. WRITE_IMAGE can write most types of image files supported by IDL.

Syntax

```
WRITE_IMAGE, Filename, Format, Data [, Red, Green, Blue] [, /APPEND]
```

Arguments

Filename

A scalar string containing the name of the file to write.

Format

A scalar string containing the name of the file format to write. See QUERY_IMAGE for a list of supported formats.

Data

An IDL variable containing the image data to write to the file.

Red

An optional vector containing the red channel of the color table if a color table exists.

Green

An optional vector containing the green channel of the color table if a color table exists.

Blue

An optional vector containing the blue channel of the color table if a color table exists.

Keywords

APPEND

Set this keyword to force the image to be appended to the file instead of overwriting the file. APPEND may be used with image formats that supports multiple images per file and is ignored for formats that support only a single image per file.

WRITE_WAV

The WRITE_WAV function writes the audio stream to the named .WAV file.

Syntax

WRITE_WAV, *Filename*, *Data*, *Rate*

Arguments

Filename

A scalar string containing the full pathname of the .WAV file to write.

Data

The array to write into the new .WAV file. The array can be a one- or two-dimensional array. A two-dimensional array is written as a multi-channel audio stream where the leading dimension of the IDL array is the number of channels. If the input array is in BYTE format, the data is written as 8-bit samples, otherwise, the data is written as signed 16-bit samples.

Rate

The sampling rate for the data array in samples per second.

Keywords

None.

XOBJVIEW

The XOBJVIEW procedure is used to quickly and easily view and manipulate IDL Object Graphics on screen. It displays given objects in an IDL widget with toolbar buttons and menus providing functionality for manipulating, printing, and exporting the resulting graphic. The mouse can be used to rotate, scale, or translate the overall model shown in a view, or to select atomic graphic objects (or model objects which have their SELECT_TARGET property set) shown in a view.

Syntax

```
XOBJVIEW, Obj [, /BLOCK] [, GROUP=widget_id] [, STATIONARY=objref(s)]  
[, XSIZE=pixels] [, YSIZE=pixels]
```

Arguments

Obj

A reference to an atomic graphics object, an IDLgrModel, or an array of such references. If *Obj* is an array, the array can contain a mixture of such references. Also, if *Obj* is an array, all object references in the array must be unique (i.e. no two references in the array can refer to the same object).

Obj is not destroyed by XOBJVIEW when XOBJVIEW is quit or killed.

Keywords

BLOCK

Set this keyword to have XMANAGER block when this application is registered. By default, BLOCK is set equal to zero, providing access to the command line if active command line processing is available. Note that setting the BLOCK keyword causes all widget applications to block, not just this application. For more information, see the documentation for the NO_BLOCK keyword to [XMANAGER](#).

GROUP

The widget ID of the widget that calls XOBJVIEW. When this ID is specified, the death of the caller results in the death of XOBJVIEW.

STATIONARY

Set this keyword to a reference to an atomic graphics object, an IDLgrModel, or an array of such references. If this keyword is an array, the array can contain a mixture

of such references. Also, if this keyword is an array, all object references in the array must be unique (i.e., no two references in the array can refer to the same object). Objects passed to XOBJVIEW via this keyword will not scale, rotate, or translate in response to mouse events. Default stationary objects are two lights. These two lights are replaced if one or more lights are supplied via this keyword. Objects specified via this keyword are not destroyed by XOBJVIEW when XOBJVIEW is quit or killed.

XSIZE

The width of the drawable area in pixels. The default is 400.

YSIZE

The height of the drawable area in pixels. The default is 400.

Using XOBJVIEW

XOBJVIEW displays a resizable top-level base with a menu, toolbar and draw widget, as shown in the following figure:

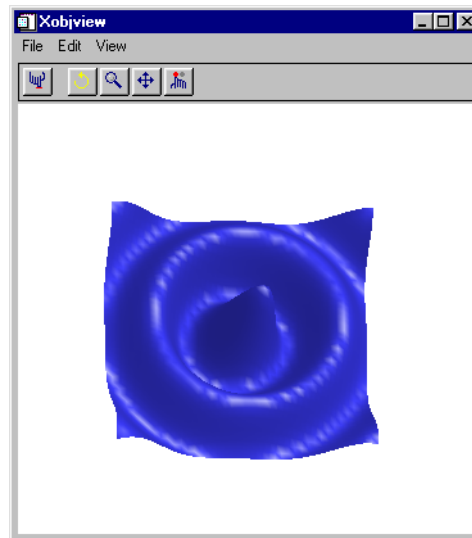


Figure 5-1: The XOBJVIEW draw widget

The XOBJVIEW Toolbar

The XOBJVIEW toolbar contains the following buttons:



Reset: Resets rotation, scaling, and panning.



Rotate: Click the left mouse button on the object and drag to rotate.



Pan: Click the left mouse button on the object and drag to pan.



Zoom: Click the left mouse button on the object and drag to zoom in or out.



Select: Click on the object. The name (or class if no name) is displayed.

Examples

Example 1

This example displays a simple IDLgrSurface object using XOBJVIEW:

```
oSurf = OBJ_NEW('IDLgrSURFACE', DIST(20))
XOBJVIEW, oSurf
```

Example 2

In this example, an IDLgrModel object consisting of two separate objects is displayed:

```
; Create contour object:
oCont = OBJ_NEW('IDLgrContour', $
    DIST(20),INDGEN(20)+20, INDGEN(20)+20, N_LEVELS=10)

; Create surface object:
oSurf = OBJ_NEW('IDLgrSurface', $
    DIST(20),INDGEN(20)+20, INDGEN(20)+20)

; Create model object:
oModel = OBJ_NEW('IDLgrModel')

; Add contour and surface objects to model:
oModel->Add, oCont
oModel->Add, oSurf

; View model:
XOBJVIEW, oModel
```

This code results in the following view in the XOBJVIEW widget:

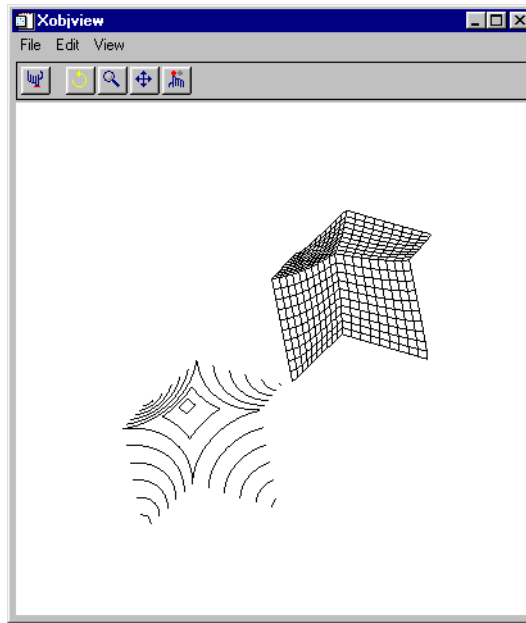


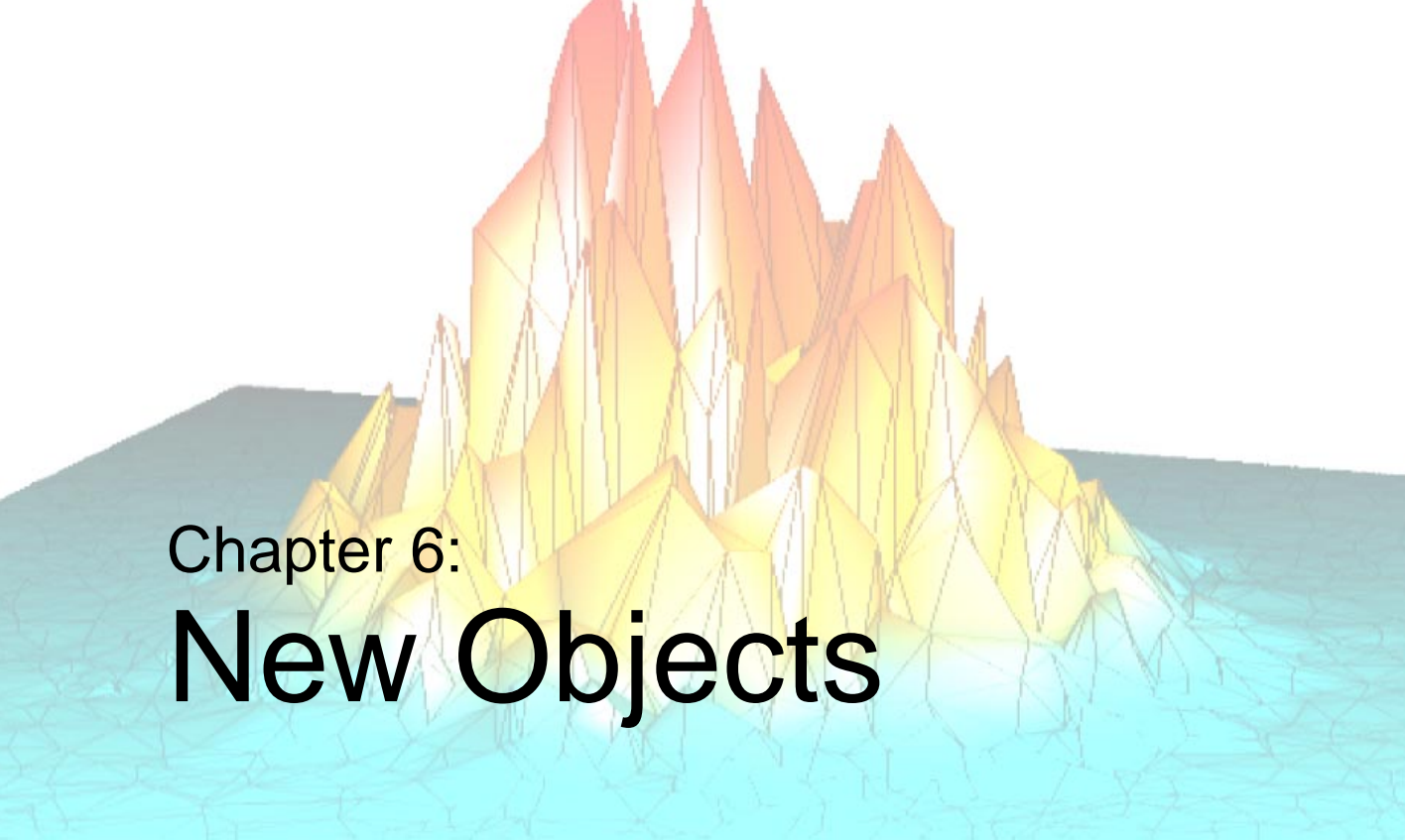
Figure 5-2: Using XOBJVIEW to view a model consisting of two objects

Note that when you click the Select button, and then click on an object, the class of that object appears next to the Select button. If you want the class of the model to appear when you click over any object in the model, you could set the `SELECT_TARGET` property of the model as follows:

```
oModel->SetProperty, /SELECT_TARGET
```

Also note that it is not necessary to create a model to view more than one object using XOBJVIEW. We could view the `oCont` and `oSurf` objects created in the above example by placing them in an array as follows:

```
XOBJVIEW, [oCont, oSurf]
```



Chapter 6: New Objects

This chapter provides documentation for IDL Objects introduced in IDL 5.3. Complete documentation for IDL Objects (including enhancements to existing objects) can be found in the *IDL Reference Guide*.

IDLanROI	284	IDLgrROI	332
IDLanROIGroup	307	IDLgrROIGroup	342
IDLffLanguageCat	324	IDLgrVRML::GetDeviceInfo	347
IDLgrBuffer::GetDeviceInfo	328	IDLgrWindow::GetDeviceInfo	349
IDLgrClipboard::GetDeviceInfo	330		

IDLanROI

The IDLanROI object class represents a region of interest.

Note

The IDLan* naming convention is used for objects in the analysis domain.

Regions of interest are described as a set of vertices that may be connected to generate a path or a polygon, or may be treated as separate points. This object may be used as a source for analytical computations on regions. (For additional information about display of ROIs in Object Graphics, refer to the [IDLgrROI](#) object class.)

Superclasses

None.

Subclasses

This class is a superclass of [IDLgrROI](#).

Creation

See [IDLanROI::Init](#).

Methods

Intrinsic Methods

The IDLanROI class has the following methods.

- [IDLanROI::AppendData](#)
- [IDLanROI::Cleanup](#)
- [IDLanROI::ComputeGeometry](#)
- [IDLanROI::ComputeMask](#)
- [IDLanROI::ContainsPoints](#)
- [IDLanROI::GetProperty](#)
- [IDLanROI::Init](#)
- [IDLanROI::RemoveData](#)

- IDLanROI::ReplaceData
- IDLanROI::Rotate
- IDLanROI::Scale
- IDLanROI::SetProperty
- IDLanROI::Translate

IDLanROI::AppendData

The IDLanROI::AppendData procedure method appends vertices to the region.

Syntax

```
Obj->[IDLanROI::]AppendData, X [, Y] [, Z] [, XRANGE=variable]  
[, YRANGE=variable] [, ZRANGE=variable]
```

Arguments

X

A vector providing the *X* components of the vertices to be appended. If the *Y* and *Z* arguments are not specified, *X* must be a two-dimensional array with the leading dimensions either 2 or 3 ([2,*] or [3,*]), in which case, *X*[0,*] represents the *X* values, *X*[1,*] represents the *Y* values, and *X*[2,*] represents the *Z* values.

Y

A vector providing the *Y* components of the vertices to be appended.

Z

A vector providing the *Z* components of the vertices to be appended.

Keywords

XRANGE

Set this keyword to a named variable that upon return contains a two-element vector, [*xmin*, *xmax*], representing the *X* range of the modification to the region. The reported range accounts for the last vertex in the region before the append occurred, as well as all vertices appended.

YRANGE

Set this keyword to a named variable that upon return contains a two-element vector, [*ymin*, *ymax*], representing the *Y* range of the modification to the region. The reported range accounts for the last vertex in the region before the append occurred, as well as all vertices appended.

ZRANGE

Set this keyword to a named variable that upon return contains a two-element vector, [*zmin*, *zmax*], representing the *Z* range of the modification to the region. The reported

range accounts for the last vertex in the region before the append occurred, as well as all vertices appended.

IDLanROI::Cleanup

The IDLanROI::Cleanup procedure method performs all cleanup for a region of interest object.

Note

Cleanup methods are special life cycle methods, and as such cannot be called outside the context of object destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Cleanup method of the subclass.

Syntax

Obj->[IDLanROI::]Cleanup

or

OBJ_DESTROY, *Obj*

(In a subclass' Cleanup method only.)

Arguments

None.

Keywords

None.

IDLanROI::ComputeGeometry

The IDLanROI::ComputeGeometry function method computes the geometrical values for area, perimeter, and/or centroid of the region.

Syntax

```
Result = Obj->[IDLanROI:]ComputeGeometry [, AREA=variable]  
[, CENTROID=variable] [, PERIMETER=variable] [, SPATIAL_OFFSET=vector]  
[, SPATIAL_SCALE=vector]
```

Return Value

Result

This function method returns a 1 for success, or a 0 for failure. Each computed value is returned in the *variable* name assigned to each keyword.

Arguments

None.

Keywords

AREA

Set this keyword to a named variable that upon return contains a floating point value representing the area of the region. Interior regions (holes) return a negative area.

CENTROID

Set this keyword to a named variable that upon return contains a floating point value representing the centroid for the region. If the TYPE of the region is 0 (points) or 1 (path), the centroid is computed as the average of each of the vertices in the region. If the TYPE of the region is 2 (polygon), the centroid is computed as a weighted average of the centroids of the polygons making up the ROI (interior centroids use negative weights). Weights are proportional to the polygon area.

PERIMETER

Set this keyword to a named variable that upon return contains a floating point value representing the perimeter of the region.

SPATIAL_OFFSET

Set this keyword to a two or three-element vector, $[tx, ty]$ or $[tx, ty, tz]$, representing the spatial calibration offset factors to be applied for the geometry calculations. The value of SPATIAL_SCALE is applied before the spatial offset values are applied. The default is $[0.0, 0.0, 0.0]$.

SPATIAL_SCALE

Set this keyword to a two or three-element vector, $[sx, sy]$ or $[sx, sy, sz]$, representing the spatial calibration scaling factors to be applied for the geometry calculations. The spatial calibration scale is applied first, then the value of SPATIAL_OFFSET is applied. The default is $[1.0, 1.0, 1.0]$.

IDLanROI::ComputeMask

The IDLanROI::ComputeMask function method prepares a two-dimensional mask for the region.

Syntax

```
Result = Obj->[IDLanROI::]ComputeMask( [, INITIALIZE={ -1 | 0 | 1 } ]
[, DIMENSIONS=[xdim, ydim] | [, MASK_IN=array] [, LOCATION=[x, y [, z]] ]
[, MASK_RULE={ 0 | 1 | 2 } ] [, PLANE_NORMAL=[x, y, z] ]
[, PLANE_XAXIS=[x,y,z] ] )
```

Return Value

Result

The return value is a two-dimensional array of bytes whose values range from 0 to 255. The mask is computed by applying the following formula to the current mask for each mask point contained within the ROI:

$$M_{out} = \text{MAX}(\text{MIN}(0, (M_{roi} * Ext) + M_{in}), 255)$$

where M_{roi} is 255 and Ext is 1 for points within an exterior region and -1 for points within an interior region.

If the TYPE of the region is 0 (points), a single mask pixel is set for each region vertex that falls within the bounds of the mask.

If the TYPE of the region is 1 (path), one-pixel-wide line segments are set within the mask.

If the TYPE of the region is 2 (closed polygon), a mask pixel is set if that pixel is on the plane of a region, and the pixel falls within the region (according to the MASK_RULE).

Arguments

None.

Keywords

DIMENSIONS

Set this keyword to a two-element vector, [*xdim*, *ydim*], specifying the requested dimensions of the returned mask. If MASK_IN is provided, the value of this keyword

is ignored and the dimensions of that mask are used. Otherwise, the default dimensions are [100, 100].

INITIALIZE

Set this keyword to indicate how the mask should be initialized. Valid values include:

- -1 = The mask is not initialized. This option is useful when updating an already existing mask. This is the default if the MASK_IN keyword is set.
- 0 = The mask is initialized so that each pixel is set to 0. This is the default if the MASK_IN keyword is not set.
- 1 = The mask is initialized so that each pixel is set to 255.

LOCATION

Set this keyword to a vector of the form [X, Y[, Z]] specifying the location of the origin of the mask. The default is [0, 0, 0].

MASK_IN

Set this keyword to a two or three-dimensional array representing a mask that is already allocated and to be updated for this region. If this keyword is provided, the data portion of this variable is grabbed and used in the returned value (an implicit NO_COPY). If this keyword is not provided, a mask is allocated by default to match the dimensions specified via the DIMENSIONS keyword.

MASK_RULE

Set this keyword to an integer specifying the rule used to determine whether a given pixel should be set within the mask. Valid values include:

- 0 = Boundary only. All pixels falling on a region's boundary are set.
- 1 = Interior only. All pixels falling within the region's boundary, but not on the boundary, are set.
- 2 = Boundary + Interior. All pixels falling on or within a region's boundary are set.

PLANE_NORMAL

Set this keyword to a three-element vector, [x, y, z], specifying the normal vector for the plane on which the mask is to be computed. The default is [0, 0, 1].

PLANE_XAXIS

Set this keyword to a three-element vector, $[x, y, z]$, specifying the direction vector along which each row of mask pixels is to be computed (starting at LOCATION). The default is $[1, 0, 0]$.

IDLanROI::ContainsPoints

The IDLanROI::ContainsPoints function method determines whether the given data coordinates are contained within the closed polygon region.

Syntax

Result = *Obj*->[IDLanROI::]ContainsPoints(*X* [, *Y* [, *Z*]])

Return Value

Result

The return value is a vector of values, one per provided point, indicating whether that point is contained. Valid values within this return vector include:

- 0 Exterior. The point lies strictly out of bounds of the ROI.
- 1 Interior. The point lies strictly inside the bounds of the ROI.
- 2 On edge. The point lies on an edge of the ROI boundary.
- 3 On vertex. The point matches a vertex of the ROI.

A point is considered to be exterior if:

- the point falls within the boundary of an interior region (hole).
- the point does not lie in the plane of the region.
- the region TYPE property is set to 0 (points) or 1 (path).

Arguments

X

A vector providing the *X* components of the points to be tested. If the *Y* and *Z* arguments are not specified, *X* must be a two-dimensional array with the leading dimension either 2 or 3 ([2,*] or [3,*]), in which case, *X*[0,*] represents the *X* values, *X*[1,*] represents the *Y* values, and *X*[2,*] represents the *Z* values.

Y

A vector providing the *Y* components of the points to be tested.

Z

A scalar or vector providing the Z component(s) of the points to be tested. If not provided, the Z components default to 0.0.

Keywords

None.

IDLanROI::GetProperty

The IDLanROI::GetProperty procedure method retrieves the value of a property or group of properties for the region.

Syntax

```
Obj->[IDLanROI::]GetProperty [, ALL=variable] [, ROI_XRANGE=variable]
[, ROI_YRANGE=variable] [, ROI_ZRANGE=variable]
```

Arguments

None.

Keywords

Any keyword to [IDLanROI::Init](#) followed by the word (*Get*) can be retrieved using IDLanROI::GetProperty. In addition, the following keywords are available:

ALL

Set this keyword to a named variable that will contain an anonymous structure containing the values of all of the properties associated with the state of this object. State information about the object includes things like block size, type, etc., but not vertex data.

Note

The fields in this structure may change in subsequent releases of IDL.

ROI_XRANGE

Set this keyword to a named variable. Upon return, ROI_XRANGE contains a two-element vector of the form [*xmin*, *xmax*] that specifies the range of X data coordinates covered by the region.

ROI_YRANGE

Set this keyword to a named variable. Upon return, ROI_YRANGE contains a two-element vector of the form [*ymin*, *ymax*] that specifies the range of Y data coordinates covered by the region.

ROI_ZRANGE

Set this keyword to a named variable. Upon return, ROI_ZRANGE contains a two-element vector of the form $[zmin, zmax]$ that specifies the range of Z data coordinates covered by the region.

IDLanROI::Init

The IDLanROI::Init function method initializes a region of interest object.

Note

Init methods are special life cycle methods, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Result = IDLanROI::Init( [X [, Y [, Z ]]] [, BLOCKSIZE{Get, Set}=vertices]
[, DATA{Get, Set}=array] [, /INTERIOR{Get, Set}] [, TYPE{Get}={ 0 | 1 | 2 } ] )
```

or

```
Obj = OBJ_NEW( 'IDLanROI' [, X [, Y [, Z ]]] )
```

(In a subclass' Init method only.)

Arguments

X

A vector providing the *X* components of the vertices for the region. If the *Y* and *Z* arguments are not specified, *X* must be a two-dimensional array with the leading dimension either 2 or 3 ([2,*] or [3,*]), in which case, *X*[0,*] represents the *X* values, *X*[1,*] represents the *Y* values, and *X*[2,*] represents the *Z* values.

Y

A vector providing the *Y* components of the vertices.

Z

A scalar or vector providing the *Z* component(s) of the vertices. If not provided, *Z* values default to 0.0.

Keywords

BLOCK_SIZE (Get, Set)

Set this keyword to the number of vertices to allocate per block as needed for the region. When additional vertices are required, an additional block is allocated. The default is 100.

DATA (Get, Set)

Set this keyword to a 2-by-*n* or 3-by-*n* array which defines the vertex data for the region. DATA is equivalent to the optional arguments, X, Y, and Z.

INTERIOR (Get, Set)

Set this keyword to mark this region as an interior region (i.e., a region treated as a hole). By default, the region is treated as an exterior region.

TYPE (Get)

Set this keyword to indicate the type of the region. The TYPE keyword determines how computational operations, such as mask generation, are performed. Valid values include:

- 0 points
- 1 path
- 2 closed polygon (the default)

IDLanROI::RemoveData

The IDLanROI::RemoveData procedure method removes vertices from the region.

Syntax

```
Obj->[IDLanROI::]RemoveData[, COUNT=vertices] [, START=index]  
[, XRANGE=variable] [, YRANGE=variable][, ZRANGE=variable]
```

Arguments

None.

Keywords

COUNT

Set this keyword to the number of vertices to remove. The default is one vertex.

START

Set this keyword to an index (into the region's current vertex list) where the removal is to begin. By default, the final vertex is removed.

XRANGE

Set this keyword to a named variable that upon return contains a two-element vector, [*xmin*, *xmax*], that represents the X range of the modification to the region. The reported range accounts for the vertex just before the removal (if any), the vertex just after the removal (if any), and the removed vertices.

YRANGE

Set this keyword to a named variable that upon return contains a two-element vector, [*ymin*, *ymax*], that represents the Y range of the modification to the region. The reported range accounts for the vertex just before the removal (if any), the vertex just after the removal (if any), and the removed vertices.

ZRANGE

Set this keyword to a named variable that upon return contains a two-element vector, [*zmin*, *zmax*], that represents the Z range of the modification to the region. The reported range accounts for the vertex just before the removal (if any), the vertex just after the removal (if any), and the removed vertices.

IDLanROI::ReplaceData

The IDLanROI::ReplaceData procedure method replaces vertices in the region with alternate values. The number of replacement values need not match the number of values being replaced.

Syntax

```
Obj->[IDLanROI::]ReplaceData, X[, Y[, Z]] [, START=index] [, FINISH=index]
[, XRANGE=variable] [, YRANGE=variable] [, ZRANGE=variable]
```

Arguments

X

A vector providing the *X* components of the new replacement vertices. If the *Y* and *Z* arguments are not specified, *X* must be a two-dimensional array with the leading dimensions either 2 or 3 ([2, *] or [3, *]), in which case, *X*[0, *] represents the *X* values, *X*[1, *] represents the *Y* values, and *X*[2, *] represents the *Z* values.

Y

A vector providing the *Y* components of the new replacement vertices.

Z

A vector providing the *Z* components of the new replacement vertices.

Keywords

FINISH

Set this keyword to the index of the region's current subregion vertex list where the replacement ends. If the START keyword value is ≥ 0 , the default FINISH is given by

$$\text{FINISH} = ((\text{START} + \text{N_NEW} - 1) \text{ MOD } \text{N_OLD})$$

where N_NEW is the number of replacement vertices provided via the [*X*, *Y*, *Z*] arguments and N_OLD is the number of vertices (prior to replacement) in the current subregion.

If the START keyword is not set or is negative, the default FINISH is given by

$$\text{FINISH} = \text{N_OLD} - 1$$

FINISH may be less than START in which case the vertices, including and following START and the vertices preceding and including FINISH, are replaced with the new values.

START

Set this keyword to an index of the region's current subregion vertex list where the replacement begins. If the FINISH keyword value is ≥ 0 , the default START is given by

$$\text{START} = ((\text{FINISH} - \text{N_NEW} + 1) \text{MOD } \text{N_OLD})$$

where N_NEW is the number of replacement vertices provided via the [X, Y, Z] arguments and N_OLD is the number of vertices (prior to replacement) in the current subregion.

If the FINISH keyword is not set (or negative), the default START is clamped to 0 and is given by

$$\text{N_OLD} - \text{N_NEW}$$

XRANGE

Set this keyword to a named variable that upon return contains a two-element vector, [*xmin*, *xmax*], representing the X range of the modification to the region. The reported range accounts for the replaced vertices, the vertex just before the replacement (if any), the vertex just after the replacement (if any), and the new replacement vertices.

YRANGE

Set this keyword to a named variable that upon return contains a two-element vector, [*ymin*, *ymax*], representing the Y range of the modification to the region. The reported range accounts for the replaced vertices, the vertex just before the replacement (if any), the vertex just after the replacement (if any), and the new replacement vertices.

ZRANGE

Set this keyword to a named variable that upon return contains a two-element vector, [*zmin*, *zmax*], representing the Z range of the modification to the region. The reported range accounts for the replaced vertices, the vertex just before the replacement (if any), the vertex just after the replacement (if any), and the new replacement vertices.

IDLanROI::Rotate

The IDLanROI::Rotate procedure method modifies the vertices for the region by applying a rotation.

Syntax

Obj→[IDLanROI::]Rotate, *Axis*, *Angle* [, CENTER=[*x*, *y*[, *z*]]]

Arguments

Axis

A three-element vector of the form [*x*, *y*, *z*] describing the axis about which the region is to be rotated.

Angle

The angle, measured in degrees, by which the rotation is to occur.

Keywords

CENTER

Set this keyword to a two or three-element vector of the form [*x*, *y*], or [*x*, *y*, *z*] specifying the center of rotation. The default is [0, 0, 0].

IDLanROI::Scale

The IDLanROI::Scale procedure method modifies the vertices for the region by applying a scale.

Syntax

Obj→[IDLanROI::]Scale, *Sx*[, *Sy*[, *Sz*]]

Arguments

Sx

The *X* scale factor. If the *Sy* and *Sz* arguments are not specified, *Sx* must be a two or three-element vector, in which case *Sx*[0] represents the scale in *X*, *Sx*[1] represents the scale in *Y*, *Sx*[2] represents the scale in *Z*.

Sy

The *Y* scale factor.

Sz

The *Z* scale factor.

Keywords

None.

IDLanROI:: SetProperty

The IDLanROI::SetProperty procedure method sets the value of a property or group of properties for the region.

Syntax

Obj->[IDLanROI::]SetProperty

Arguments

None.

Keywords

Any keywords to [IDLanROI::Init](#) followed by the word (*Set*) can be set using IDLanROI::SetProperty.

IDLanROI::Translate

The IDLanROI::Translate procedure method modifies the vertices for the region by applying a translation.

Syntax

Obj→[IDLanROI::]Translate, *Tx*[, *Ty*[, *Tz*]]

Arguments

Tx

The *X* translation factor. If the *Ty* and *Tz* arguments are not specified, *Tx* must be a two or three-element vector, in which case *Tx*[0] represents translation in *X*, *Tx*[1] represents translation in *Y*, *Tx*[2] represents translation in *Z*.

Ty

The *Y* translation factor.

Tz

The *Z* translation factor.

Keywords

None.

IDLanROIGroup

The IDLanROIGroup object class is an analytical representation of a group of regions of interest.

Superclasses

This class is a subclass of [IDL_Container](#).

Subclasses

This class is a superclass of [IDLgrROIGroup](#).

Creation

See [IDLanROIGroup::Init](#).

Methods

Intrinsic Methods

The IDLanROIGroup class has the following methods:

- [IDLanROIGroup::Add](#)
- [IDLanROIGroup::Cleanup](#)
- [IDLanROIGroup::ContainsPoints](#)
- [IDLanROIGroup::ComputeMask](#)
- [IDLanROIGroup::ComputeMesh](#)
- [IDLanROIGroup::GetProperty](#)
- [IDLanROIGroup::Init](#)
- [IDLanROIGroup::Rotate](#)
- [IDLanROIGroup::Scale](#)
- [IDLanROIGroup::Translate](#)

Inherited Methods

This class inherits the following methods:

- [IDL_Container::Count](#)

- `IDL_Container::Get`
- `IDL_Container::IsContained`
- `IDL_Container::Move`
- `IDL_Container::Remove`

IDLanROIGroup::Add

The IDLanROIGroup::Add procedure method adds a region to the region group. Only objects of the IDLanROI class may be added to the group. The regions in the group must all be of the same type: all points, all paths, or all polygons.

Syntax

Obj→[IDLanROIGroup::]Add, *ROI*

Arguments

ROI

A reference to an instance of the IDLanROI object class representing the region of interest to be added to the group.

Keywords

Accepts all keywords accepted by the [IDL_Container::Add](#) method in the *IDL Reference Guide*.

IDLanROIGroup::Cleanup

The IDLanROIGroup::Cleanup procedure method performs all cleanup for a region of interest group object.

Note

Cleanup methods are special life cycle methods, and as such cannot be called outside the context of object destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Cleanup method of the subclass.

Syntax

Obj→[IDLanROIGroup::]Cleanup

or

OBJ_DESTROY, *Obj*

(In a subclass' Cleanup method only.)

Arguments

None.

Keywords

None.

IDLanROIGroup::ContainsPoints

The IDLanROIGroup::ContainsPoints procedure method determines whether the given points (in data coordinates) are contained within the closed polygon regions within this group.

A point is considered to be exterior if any of the following conditions are true:

- the point falls within the boundary of an interior region (hole).
- the point does not lie in the plane of any of the contained regions.
- the TYPE property of the contained regions is set to 0 (points) or 1 (path).

Syntax

```
Result = Obj->[IDLanROIGroup::]ContainsPoints( X[, Y[, Z]] )
```

Return Value

Result

The return value is a vector of values, one per provided point, indicating whether that point is contained. Valid values within this return vector include:

- 0 Exterior. The point lies strictly outside the bounds of the ROI.
- 1 Interior. The point lies strictly inside the bounds of the ROI.
- 2 On Edge. The point lies on an edge of the ROI boundary.
- 3 On Vertex. The point matches a vertex of the ROI.

Arguments

X

A vector providing the *X* components of the points to be tested. If the *Y* and *Z* arguments are not specified, *X* must be a two-dimensional array with the leading dimension either 2 or 3 ([2,*] or [3,*]), in which case, *X*[0,*] represents the *X* values, *X*[1,*] represents the *Y* values, and *X*[2,*] represents the *Z* values.

Y

A vector providing the *Y* components of the points to be tested.

Z

A scalar or vector providing the *Z* components of the points to be tested. If not provided, the *Z* components default to 0.0.

Keywords

None.

IDLanROIGroup::ComputeMask

The IDLanROIGroup::ComputeMask function method prepares a two-dimensional mask for this group of regions.

Syntax

```
Result = Obj->[IDLanROIGroup::]ComputeMask( [, INITIALIZE={ -1 | 0 | 1 }]
[, DIMENSIONS=[xdim, ydim] | [, MASK_IN=array] [, LOCATION=[x, y [, z]]]
[, MASK_RULE={ 0 | 1 | 2 }])
```

Return Value

Result

The return value is a two-dimensional array of bytes whose values range from 0 to 255. The mask is computed by applying the following formula to the current mask for each mask point contained within the ROI:

$$M_{out} = \text{MAX}(\text{MIN}(0, (M_{roi} * Ext) + M_{in}), 255)$$

where M_{roi} is 255 and Ext is 1 for points within an exterior region and -1 for points within an interior region.

If the TYPE of the contained regions is 0 (points), a single mask pixel is set for each region vertex that falls within the bounds of the mask.

If the TYPE of the contained regions is 1 (path), each pixel along the paths of the regions is set if it falls within the mask.

If the TYPE of the region is 2 (closed polygon), a mask pixel is set if that pixel is on the plane of a contained region, and the pixel falls within that region (according to the MASK_RULE).

Arguments

None.

Keywords

DIMENSIONS

Set this keyword to a two-element vector, [*xdim*, *ydim*], specifying the requested dimensions of the returned mask. If MASK_IN is provided, the value of this keyword

is ignored, and the dimensions of that mask are used. Otherwise, the default dimensions are [100, 100].

INITIALIZE

Set this keyword to indicate how the mask should be initialized. Valid values include:

- 1 The mask is not initialized; the default if the `MASK_IN` keyword is set. This option is useful when updating an already existing mask.
- 0 The mask is initialized with each pixel set to 0; the default if the `MASK_IN` keyword is not set.
- 1 The mask is initialized with each pixel set to 255.

LOCATION

Set this keyword to a vector of the form [X, Y[, Z]] specifying the location of the origin of the mask. The default is [0, 0, 0].

MASK_IN

Set this keyword to a two or three-dimensional array representing a mask that is already allocated and to be updated for this region. If this keyword is provided, the data portion of this variable is grabbed and used in the returned value (an implicit `NO_COPY`). If this keyword is not provided, a mask is allocated by default to match the dimensions specified via the `DIMENSIONS` keyword.

MASK_RULE

Set this keyword to an integer specifying the rule used to determine whether a given pixel should be set within the mask. Valid values include:

- 0 Boundary Only. All pixels falling on a region's boundary are set.
- 1 Interior Only. All pixels falling within the region's boundary, but not on the boundary, are set.
- 2 Boundary + Interior. All pixels falling on or within a region's boundary are set.

PLANE_NORMAL

Set this keyword to a three-element vector, $[x, y, z]$, specifying the normal vector for the plane on which the mask is to be computed. The default is $[0, 0, 1]$.

PLANE_XAXIS

Set this keyword to a three-element vector, $[x, y, z]$, specifying the direction vector along which each row of mask pixels is to be computed (starting at LOCATION). The default is $[1, 0, 0]$.

IDLanROIGroup::ComputeMesh

The IDLanROIGroup::ComputeMesh function method triangulates a surface mesh with optional capping from the stack of regions contained within this group.

Note

The contained regions may be concave. However, this method will fail under the following conditions:

- The region group contains fewer than two regions.
- The TYPE property of the contained regions is 0 (points) or 1 (path).
- Any of the contained regions are not simple (i.e., a region is self-intersecting).
- The region group contains interior regions (holes).
- More than one region lies on the same plane (i.e., the region group contains branches).

Each region pair is normalized by perimeter and the triangulation is computed by walking the contours in parallel, keeping the normalized progress along each contour in sync. The returned triangulation minimizes the mesh surface area. Each vertex may appear only once in the output, and the resulting polygon mesh is solid with outward facing normals computed via the right-hand rule. If capping is requested, it is computed using the [IDLgrTessellator](#) on the top and bottom regions, and/or the regions on either side of an inter-slice gap.

Syntax

```
Result = Obj->[IDLanROIGroup:]ComputeMesh( Vertices, Conn
[, CAPPED={ 0 | 1 | 2}] [, SURFACE_AREA=variable] )
```

Return Value

Result

The return value of this function method is the number of triangles generated if the surface mesh triangulation is successful, or zero if unsuccessful.

Arguments

Vertices

An output [3, *n*] array of float vertices.

Conn

An output polygon mesh connectivity array.

Keywords

CAPPED

Set this keyword to a value to indicate whether flat caps are to be computed at the top-most or bottom-most regions (as selected by a counter-clockwise rule), or at the regions on either side of an inter-slice gap. The value of this keyword is a bit-wise OR of the values shown below. For example, to cap the top-most and bottom-most regions only, set the CAPPED keyword to 3. The default is 0 (no caps).

- 0 no caps
- 1 cap the top-most region
- 2 cap the bottom-most region

SURFACE_AREA

Set this keyword to a named variable that upon return contains the overall surface area of the computed triangulation. This value was minimized in the computation of the triangulation.

IDLanROIGroup::GetProperty

The IDLanROIGroup::GetProperty procedure method retrieves the value of a property or group of properties for the region group.

Syntax

```
Obj->[IDLanROIGroup::]GetProperty[, ALL=variable]  
[, ROIGROUP_XRANGE=variable] [, ROIGROUP_YRANGE=variable]  
[, ROIGROUP_ZRANGE=variable]
```

Arguments

None.

Keywords

Any keyword to [IDLanROIGroup::Init](#) followed by the word (*Get*) can be retrieved using IDLanROIGroup::GetProperty. In addition, the following keywords are available:

ALL

Set this keyword to a named variable. Upon return, ALL contains an anonymous structure with the values of all of the properties associated with the state of this object.

Note

The fields in this structure may change in subsequent releases of IDL.

ROIGROUP_XRANGE

Set this keyword to a named variable. Upon return, ROIGROUP_XRANGE contains a two-element vector of the form [*xmin*, *xmax*] specifying the range of X data coordinates covered by the regions in this group.

ROIGROUP_YRANGE

Set this keyword to a named variable. Upon return, ROIGROUP_YRANGE contains a two-element vector of the form [*ymin*, *ymax*] specifying the range of Y data coordinates covered by the regions in this group.

ROIGROUP_ZRANGE

Set this keyword to a named variable. Upon return, ROIGROUP_ZRANGE contains a two-element vector of the form $[zmin, zmax]$ specifying the range of Z data coordinates covered by the regions in this group.

IDLanROIGroup::Init

The IDLanROIGroup::Init function method initializes a region of interest group object.

Note

Init methods are special life cycle methods, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

Result = IDLanROIGroup::Init()

or

Obj = OBJ_NEW('IDLanROIGroup')

(In a subclass' Init method only.)

Arguments

None.

Keywords

None.

IDLanROIGroup::Rotate

The IDLanROIGroup::Rotate procedure method modifies the vertices for all regions within the group by applying a rotation.

Syntax

Obj→[IDLanROIGroup:]Rotate, *Axis*, *Angle*[, CENTER=[*x*, *y* [, *z*]]]

Arguments

Axis

A three-element vector of the form [*x*, *y*, *z*] describing the axis about which the region group is to be rotated.

Angle

The angle, measured in degrees, by which to rotate the ROI group.

Keywords

CENTER

Set this keyword to a two or three-element vector of the form [*x*, *y*] or [*x*, *y*, *z*] specifying the center of rotation. The default is [0, 0, 0].

IDLanROIGroup::Scale

The IDLanROIGroup::Scale procedure method modifies the vertices for the region by applying a scale.

Syntax

Obj→[IDLanROIGroup::]Scale, *Sx*[, *Sy*[, *Sz*]]

Arguments

Sx

The *X* scale factor. If the *Sy* and *Sz* arguments are not specified, *Sx* must be a two or three-element vector, in which case *Sx*[0] represents the scale in *X*, *Sx*[1] represents the scale in *Y*, and *Sx*[2] represents the scale in *Z*.

Sy

The *Y* scale factor.

Sz

The *Z* scale factor.

Keywords

None.

IDLanROIGroup::Translate

The IDLanROIGroup::Translate procedure method modifies the vertices of all regions within the group by applying a translation.

Syntax

Obj→[IDLanROIGroup:]Translate, *Tx*[, *Ty*[, *Tz*]]

Arguments

Tx

The *X* translation factor. If the *Ty* and *Tz* arguments are not specified, *Tx* must be a two or three-element vector, in which case *Tx*[0] represents translation in *X*, *Tx*[1] represents translation in *Y*, and *Tx*[2] represents translation in *Z*.

Ty

The *Y* translation factor.

Tz

The *Z* translation factor.

Keywords

None.

IDLffLanguageCat

The IDLffLanguageCat object provides an interface to IDL language catalog files.

Note

This object is not savable. Restored IDLffLanguageCat objects may contain invalid data.

Note

This object is not intended to be created with OBJ_NEW. The [MSG_CAT_OPEN](#) function is used to return the correct object reference.

Superclasses

This class has no superclasses.

Subclasses

This class has no subclasses.

Creation

See [MSG_CAT_OPEN](#).

Methods

- [IDLffLanguageCat::IsValid](#)
- [IDLffLanguageCat::Query](#)
- [IDLffLanguageCat::SetCatalog](#)

IDLffLanguageCat::IsValid

The IDLffLanguageCat::IsValid function method is used to determine whether the object has a valid catalog.

Syntax

Result = Obj ->[IDLffLanguageCat:]IsValid()

Arguments

None

Keywords

None

IDLffLanguageCat::Query

The IDLffLanguageCatalog::Query function method is used to return the language string associated with the given key. If the key is not found in the given catalog, the default string is returned.

Syntax

```
Result = Obj ->[IDLffLanguageCat::]Query( Key [, DEFAULT_STRING=string] )
```

Arguments

Key

The scalar, or array of (string) keys associated with the desired language string. If key is an array, *Result* will be a string array of the associated language strings.

Keywords

DEFAULT_STRING

Set this keyword to the desired value of the return string if the key cannot be found in the catalog file. The default value is the empty string.

IDLffLanguageCat::SetCatalog

The IDLffLanguageCat::SetCatalog function method is used to set the appropriate catalog file. This function returns 1 upon success, and 0 on failure.

Syntax

```
Result = Obj ->[IDLffLanguageCat:]SetCatalog( Application  
[, FILENAME=string] [, LOCALE=string] [, PATH=string] )
```

Arguments

Application

A scalar string representing the name of the desired application's catalog file.

Keywords

FILENAME

Set this keyword to a scalar string containing the full path and filename of the catalog file to open. If this keyword is set, *application*, PATH, and LOCALE are ignored.

LOCALE

Set this keyword to the desired locale for the catalog file. If not set, the current locale is used.

PATH

Set this keyword to a scalar string containing the path to search for language catalog files. The default is the current directory.

IDLgrBuffer::GetDeviceInfo

The IDLgrBuffer::GetDeviceInfo function method returns information which allows IDL applications to intelligently make decisions for optimal performance. For example, it allows an application to determine if RENDERER=1 is actually implemented in hardware. It also allows applications to make optimal quality decisions when dynamically building texture maps.

Syntax

```
Result = Obj->[IDLgrBuffer::]GetDeviceInfo( [, ALL=variable]  
[, MAX_TEXTURE_DIMENSIONS=variable]  
[, MAX_VIEWPORT_DIMENSIONS=variable] [, NAME=variable]  
[, NUM_CPUS=variable] [, VENDOR=variable] [, VERSION=variable] )
```

Arguments

None.

Keywords

ALL

Set this keyword to a named variable which, upon return, contains a structure with the values of all the device information keywords as fields.

MAX_TEXTURE_DIMENSIONS

Set this keyword equal to a named variable. Upon return, MAX_TEXTURE_DIMENSIONS contains a two-element integer array that specifies the maximum texture size supported by the device.

MAX_VIEWPORT_DIMENSIONS

Set this keyword equal to a named variable. Upon return, MAX_VIEWPORT_DIMENSIONS contains a two-element integer array that specifies the maximum size of a graphics display supported by the device.

NAME

Set this keyword equal to a named variable. Upon return, NAME contains the name of the rendering device as a string.

NUM_CPUS

Set this keyword equal to a named variable. Upon return, NUM_CPUS contains an integer that specifies the number of CPUs that are known to, and available to IDL.

Note

The NUM_CPUS keyword accurately returns the number of CPUs for the SGI Irix, SUN, and Microsoft Windows platforms. For platforms other than these, the number returned may not reflect the actual number of CPUs available to IDL in the current system.

VENDOR

Set this keyword equal to a named variable. Upon return, VENDOR contains the name of the rendering device creator as a string.

VERSION

Set this keyword equal to a named variable. Upon return, VERSION contains the version of the rendering device driver as a string.

IDLgrClipboard::GetDeviceInfo

The IDLgrClipboard::GetDeviceInfo function method returns information which allows IDL applications to intelligently make decisions for optimal performance. For example, it allows an application to determine if RENDERER=1 is actually implemented in hardware. It also allows applications to make optimal quality decisions when dynamically building texture maps.

Syntax

```
Result = Obj->[IDLgrClipboard::]GetDeviceInfo( [, ALL=variable]  
[, MAX_TEXTURE_DIMENSIONS=variable]  
[, MAX_VIEWPORT_DIMENSIONS=variable] [, NAME=variable]  
[, NUM_CPUS=variable] [, VENDOR=variable] [, VERSION=variable] )
```

Arguments

None.

Keywords

ALL

Set this keyword to a named variable which, upon return, contains a structure with the values of all the device information keywords as fields.

MAX_TEXTURE_DIMENSIONS

Set this keyword equal to a named variable. Upon return, MAX_TEXTURE_DIMENSIONS contains a two element integer array that specifies the maximum texture size supported by the device.

MAX_VIEWPORT_DIMENSIONS

Set this keyword equal to a named variable. Upon return, MAX_VIEWPORT_DIMENSIONS contains a two element integer array that specifies the maximum size of a graphics display supported by the device.

NAME

Set this keyword equal to a named variable. Upon return, NAME contains the name of the rendering device as a string.

NUM_CPUS

Set this keyword equal to a named variable. Upon return, NUM_CPUS contains an integer that specifies the number of CPUs that are known to, and available to IDL.

Note

The NUM_CPUS keyword accurately returns the number of CPUs for the SGI Irix, SUN, and Microsoft Windows platforms. For platforms other than these, the number returned may not reflect the actual number of CPUs available to IDL in the current system.

VENDOR

Set this keyword equal to a named variable. Upon return, VENDOR contains the name of the rendering device creator as a string.

VERSION

Set this keyword equal to a named variable. Upon return, VERSION contains the version of the rendering device driver as a string.

IDLgrROI

The IDLgrROI object class is an object graphics representation of a region of interest.

Superclasses

This class is a subclass of [IDLanROI](#).

Subclasses

None.

Creation

See [IDLgrROI::Init](#).

Methods

Intrinsic Methods

The IDLgrROI object class has the following methods:

- [IDLgrROI::Cleanup](#)
- [IDLgrROI::GetProperty](#)
- [IDLgrROI::Init](#)
- [IDLgrROI::PickVertex](#)
- [IDLgrROI::SetProperty](#)

Inherited Methods

This class inherits the following methods:

- [IDLanROI::AppendData](#)
- [IDLanROI::ComputeGeometry](#)
- [IDLanROI::ComputeMask](#)
- [IDLanROI::ContainsPoints](#)
- [IDLanROI::RemoveData](#)
- [IDLanROI::ReplaceData](#)

- [IDLanROI::Rotate](#)
- [IDLanROI::Scale](#)
- [IDLanROI::Translate](#)

IDLgrROI::Cleanup

The IDLgrROI::Cleanup procedure method performs all cleanup for a region of interest object.

Note

Cleanup methods are special life cycle methods, and as such cannot be called outside the context of object destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Cleanup method of the subclass.

Syntax

Obj->[IDLgrROI::]Cleanup

or

OBJ_DESTROY, *Obj*

(In a subclass' Cleanup method only.)

Arguments

None.

Keywords

None.

IDLgrROI::GetProperty

The IDLgrROI::GetProperty procedure method retrieves the value of a property or group of properties for the Object Graphics region.

Syntax

Obj→[IDLgrROI:]GetProperty [, ALL=*variable*]

Arguments

None.

Keywords

Note

All keywords accepted by [IDLanROI::GetProperty](#) are also accepted by this method. Furthermore, any keyword to [IDLgrROI::Init](#) followed by the word (*Get*) can be retrieved using IDLgrROI::GetProperty.

The following keywords are also accepted:

ALL

Set this keyword to a named variable to contain an anonymous structure with the values of all of the properties associated with the state of this object. State information about the object may include things like color, line style, etc., but not vertex data or user values.

Note

The fields in this structure may change in subsequent releases of IDL.

IDLgrROI::Init

The IDLgrROI::Init function method initializes an Object Graphics region of interest.

Note

Init methods are special life cycle methods, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Result = IDLgrROI::Init([X[, Y[, Z]]) [, COLOR{Get, Set}=vector]
[, /HIDE{Get, Set}] [, LINESTYLE{Get, Set}=value] [, NAME{Get, Set}=string]
[, PALETTE{Get, Set}=objref] [, STYLE{Get, Set}={ 0 | 1 | 2 } ]
[, SYMBOL{Get, Set}=objref] [, THICK{Get, Set}=points {1.0 to 10.0} ]
[, UVALUE{Get, Set}=uvalue] [, XCOORD_CONV{Get, Set}=[s0, s1] ]
[, YCOORD_CONV{Get, Set}=[s0, s1] ] [, ZCOORD_CONV{Get, Set}=[s0, s1] ] )
```

or

```
Obj = OBJ_NEW( 'IDLgrROI'[, X[, Y[, Z]]) )
```

(In a subclass' Init method only.)

Arguments

X

A vector providing the *X* components of the vertices for the region. If the *Y* and *Z* arguments are not specified, *X* must be a two-dimensional array with the leading dimension either 2 or 3 ([2, *] or [3, *]), in which case, *X*[0, *] represents the *X* values, *X*[1, *] represents the *Y* values, and *X*[2, *] represents the *Z* values.

Y

A vector providing the *Y* components of the vertices.

Z

A scalar or vector providing the *Z* components of the vertices. If not provided, *Z* values default to 0.0.

Keywords

Note

All keywords accepted by `IDLanROI::Init` are accepted by this method as well.

In addition, the following keywords are accepted:

COLOR (Get, Set)

Set this keyword to an RGB or indexed color for drawing the region. The default color is [0, 0, 0].

HIDE (Get, Set)

Set this keyword to a Boolean value indicating whether this region should be drawn:

- 0 draw the region (the default)
- 1 do not draw the region

LINestyle (Get, Set)

Set this keyword to the line style to be used to draw the region. The value can be either an integer value specifying a pre-defined line style, or a two-element vector specifying a stippling pattern.

The valid values for the pre-defined line styles are:

- 0 solid (the default)
- 1 dotted
- 2 dashed
- 3 dash dot
- 4 dash dot dot dot
- 5 long dash
- 6 no line drawn

NAME (Get, Set)

Set this keyword to a string to use as the name for this region.

PALETTE (Get, Set)

Set this keyword to the object reference of a palette object (an instance of the [IDLgrPalette](#) object class). This keyword is only used for Object Graphics destinations using the RGB color model. In this case, if the color value for the region is specified as a color index value, this palette is used to look up the color for the region. If the PALETTE keyword is not set, the destination object PALETTE property is used, which defaults to a gray scale ramp.

STYLE (Get, Set)

Set this keyword to indicate the geometrical primitive to use to represent the region when displayed. Valid values include:

- 0 points
- 1 open polyline
- 2 closed polyline (the default)

SYMBOL (Get, Set)

Set this keyword to reference an [IDLgrSymbol](#) object for the symbol used for display when STYLE = 0 (points). By default, a dot is used.

THICK (Get, Set)

Set this keyword to a value between 1.0 and 10.0, specifying the size of the points, or the thickness of the lines, measured in points. The default is one point.

UVALUE (Get, Set)

Set this keyword to a user value of any type to contain any information you wish. Remember if you set this user value equal to a pointer or object reference, you must destroy the pointer or object reference explicitly when destroying this region.

XCOORD_CONV (Get, Set)

Set this keyword to a vector, $[s_0, s_1]$, of scaling factors used to convert X coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedX} = s_0 + s_1 * \text{DataX}$$

Recommended values are:

$$[(-X_{min}) / (X_{max} - X_{min}), 1.0 / (X_{max} - X_{min})]$$

YCOORD_CONV (Get, Set)

Set this keyword to a vector, $[s_0, s_1]$, of scaling factors used to convert Y coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{Normalized}Y = s_0 + s_1 * \text{Data}Y$$

Recommended values are:

$$[(-Y_{min}) / (Y_{max} - Y_{min}), 1.0 / (Y_{max} - Y_{min})]$$

ZCOORD_CONV (Get, Set)

Set this keyword to a vector, $[s_0, s_1]$, of scaling factors used to convert Z coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{Normalized}Z = s_0 + s_1 * \text{Data}Z$$

Recommended values are:

$$[(-Z_{min}) / (Z_{max} - Z_{min}), 1.0 / (Z_{max} - Z_{min})]$$

IDLgrROI::PickVertex

The IDLgrROI::PickVertex function method picks a vertex of the region which, when projected onto the given destination device, is nearest to the given 2D device coordinate.

Syntax

```
Result = Obj->[IDLgrROI::]PickVertex( Dest, View, Point [, PATH=objref] )
```

Return Value

Result

The return value is the index of the nearest region vertex. If two or more vertices are equally nearest to the point, the smallest index of those vertices is returned.

Arguments

Dest

An object reference to an [IDLgrWindow](#) or [IDLgrBuffer](#) for which the pick is to occur.

View

An object reference to the [IDLgrView](#) containing this region.

Point

A two-element vector, [x, y], representing the device location used for picking a nearest vertex.

Keywords

PATH

Set this keyword to a single object reference or a vector of object references. This keyword specifies the path in the graphics hierarchy to map the device position to a location in the data space of the region. Each path object reference specified with this keyword must contain an alias. The selected vertex is computed for the version of the object falling within the specified path. If this keyword is not set, the parent properties determine the path from the current object to the top of the graphics hierarchy and no alias paths are pursued.

IDLgrROI:: SetProperty

The IDLgrROI::SetProperty procedure method sets the value of a property or group of properties for the Object Graphics region.

Syntax

Obj→[IDLgrROI::]SetProperty

Arguments

None.

Keywords

Note

Any keywords accepted by [IDLanROI::SetProperty](#) are also accepted by this method. Furthermore, any keywords to [IDLgrROI::Init](#) followed by the word (*Set*) can be set using IDLgrROI::SetProperty as well.

IDLgrROIGroup

The IDLgrROIGroup object class is an Object Graphics representation of a group of regions of interest.

Superclasses

This class is a subclass of [IDLanROIGroup](#).

Subclasses

None.

Creation

See [IDLgrROIGroup::Init](#).

Methods

Intrinsic Methods

The IDLgrROIGroup class has the following methods:

- [IDLgrROIGroup::Add](#)
- [IDLgrROIGroup::Cleanup](#)
- [IDLgrROIGroup::Init](#)
- [IDLgrROIGroup::PickRegion](#)

Inherited Methods

This class inherits the following methods:

- [IDLanROIGroup::ContainsPoints](#)
- [IDLanROIGroup::ComputeMask](#)
- [IDLanROIGroup::GetProperty](#)
- [IDLanROIGroup::Rotate](#)
- [IDLanROIGroup::Scale](#)
- [IDLanROIGroup::Translate](#)

IDLgrROIGroup::Add

The IDLgrROIGroup::Add procedure method adds a region to the region group. Only objects of the IDLgrROI class may be added to the group. The regions in the group must all be of the same type: all points, all paths, or all polygons.

Syntax

Obj→[IDLgrROIGroup::]Add, *ROI*

Arguments

ROI

A reference to an instance of the IDLgrROI object class representing the region of interest to add to the group.

Keywords

Accepts all keywords accepted by the [IDLanROIGroup::Add](#) method.

IDLgrROIGroup::Cleanup

The IDLgrROIGroup::Cleanup procedure method performs all cleanup for an Object Graphics region of interest group object.

Note

Cleanup methods are special life cycle methods, and as such cannot be called outside the context of object destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Cleanup method of the subclass.

Syntax

Obj->[IDLgrROIGroup::]Cleanup

or

OBJ_DESTROY, *Obj*

(In a subclass' Cleanup method only.)

Arguments

None.

Keywords

None.

IDLgrROIGroup::Init

The IDLgrROIGroup::Init function method initializes an Object Graphics region of interest group object.

Note

Init methods are special life cycle methods, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

Result = IDLgrROIGroup::Init()

or

Obj = OBJ_NEW('IDLgrROIGroup')

(In a subclass' Init method only.)

Arguments

None.

Keywords

None.

IDLgrROIGroup::PickRegion

The IDLgrROIGroup::PickRegion function method picks a region within the group which, when projected onto the given destination device, is nearest to the given 2D device coordinate.

Syntax

Result = Obj->[IDLgrROIGroup:]PickRegion(Dest, View, Point [, PATH=objref])

Return Value

Result

The return value is the object reference of the nearest region. If two or more regions are equally nearest to the point, the one that was added to the region group first is returned.

Arguments

Dest

An object reference to an [IDLgrWindow](#) or [IDLgrBuffer](#) for which the pick is to occur.

View

An object reference to the [IDLgrView](#) containing this region.

Point

A two-element vector, [x, y], representing the device location to use for picking a nearest region.

Keywords

PATH

Set this keyword to a single object reference or a vector of object references. This keyword specifies the path in the graphics hierarchy to map the device position to a location in the data space of the region. Each path object reference specified with this keyword must contain an alias. The selected region is computed for the version of the object falling within the specified path. If this keyword is not set, the parent properties determine the path from the current object to the top of the graphics hierarchy and no alias paths are pursued.

IDLgrVRML::GetDeviceInfo

The IDLgrVRML::GetDeviceInfo function method returns information which allows IDL applications to intelligently make decisions for optimal performance. For example, it allows an application to determine if RENDERER=1 is actually implemented in hardware. It also allows applications to make optimal quality decisions when dynamically building texture maps.

Syntax

```
Result = Obj->[IDLgrVRML::]GetDeviceInfo( [, ALL=variable]  
[, MAX_TEXTURE_DIMENSIONS=variable]  
[, MAX_VIEWPORT_DIMENSIONS=variable] [, NAME=variable]  
[, NUM_CPUS=variable] [, VENDOR=variable] [, VERSION=variable] )
```

Arguments

None.

Keywords

ALL

Set this keyword to a named variable which, upon return, contains a structure with the values of all the device information keywords as fields.

MAX_TEXTURE_DIMENSIONS

Set this keyword equal to a named variable. Upon return, *MAX_TEXTURE_DIMENSIONS* contains a two element integer array that specifies the maximum texture size supported by the device.

MAX_VIEWPORT_DIMENSIONS

Set this keyword equal to a named variable. Upon return, *MAX_VIEWPORT_DIMENSIONS* contains a two element integer array that specifies the maximum size of a graphics display supported by the device.

NAME

Set this keyword equal to a named variable. Upon return, *NAME* contains the name of the rendering device as a string.

NUM_CPUS

Set this keyword equal to a named variable. Upon return, *NUM_CPUS* contains an integer that specifies the number of CPUs that are known to, and available to IDL.

Note

The *NUM_CPUS* keyword accurately returns the number of CPUs for the SGI Irix, SUN, and Microsoft Windows platforms. For platforms other than these, the number returned may not reflect the actual number of CPUs available to IDL in the current system.

VENDOR

Set this keyword equal to a named variable. Upon return, *VENDOR* contains the name of the rendering device creator as a string.

VERSION

Set this keyword equal to a named variable. Upon return, *VERSION* contains the version of the rendering device driver as a string.

IDLgrWindow::GetDeviceInfo

The IDLgrWindow::GetDeviceInfo procedure method returns information which allows IDL applications to intelligently make decisions for optimal performance. For example, it allows an application to determine if RENDERER=0 is actually implemented in hardware. It also allows applications to make optimal quality decisions when dynamically building texture maps.

Syntax

```
Obj->[IDLgrWindow::]GetDeviceInfo [, ALL=variable]  
[, MAX_TEXTURE_DIMENSIONS=variable]  
[, MAX_VIEWPORT_DIMENSIONS=variable] [, NAME=variable]  
[, NUM_CPUS=variable] [, VENDOR=variable] [, VERSION=variable]
```

Arguments

None.

Keywords

ALL

Set this keyword to a named variable which, upon return, contains a structure with the values of all the device information keywords as fields.

MAX_TEXTURE_DIMENSIONS

Set this keyword equal to a named variable. Upon return, MAX_TEXTURE_DIMENSIONS contains a two element integer array that specifies the maximum texture size supported by the device.

MAX_VIEWPORT_DIMENSIONS

Set this keyword equal to a named variable. Upon return, MAX_VIEWPORT_DIMENSIONS contains a two element integer array that specifies the maximum size of a graphics display supported by the device.

NAME

Set this keyword equal to a named variable. Upon return, NAME contains the name of the rendering device as a string.

NUM_CPUS

Set this keyword equal to a named variable. Upon return, NUM_CPUS contains an integer that specifies the number of CPUs that are known to, and available to IDL.

Note

The NUM_CPUS keyword accurately returns the number of CPUs for the SGI Irix, SUN, and Microsoft Windows platforms. For platforms other than these, the number returned may not reflect the actual number of CPUs available to IDL in the current system.

VENDOR

Set this keyword equal to a named variable. Upon return, VENDOR contains the name of the rendering device creator as a string.

VERSION

Set this keyword equal to a named variable. Upon return, VERSION contains the version of the rendering device driver as a string.



Index

Symbols

- `%?`, 117
- `.FULL_RESET_SESSION` command, 142
- `.prc` file
 - testing in a project, 94
- `.prj` files, 87
- `.RESET_SESSION` command, 143

Numerics

- 3D visualization
 - improvements, 13

A

- `ADAPT_HIST_EQUAL` function, 145

Add method

- `IDLAnROIGroup`, 309
- `IDLgrROIGroup`, 343
- adding
 - files to a project, 90
- Adobe Acrobat Portable Document Format (PDF), 46
- analysis objects
 - `IDLAnRIOGroup`, 307
 - `IDLAnROI` class, 284
- `AppendData` method
 - `IDLAnROI`, 286
- ASCII
 - importing using macros, 125
- audio file support, 26
- axis label
 - orientation of text, 18

B

BINARY, 25

binary data

- importing using macros, 131

BINARY_TEMPLATE function, 147

breakpoint enhancements, 110

building

- a project, 102

- order in project, 99

byte ordering

- big endian, 132

- binary data, 132

- little endian, 132

- native method, 132

C

CALL, 182, 182

CDF, 149

CDF compression, 31

CDF version, 32

CDF_COMPRESSION function, 149

Cleanup method

- IDLanROI**, 288

- IDLanROI**Group, 310

- IDLgrROI**, 334

- IDLgrROI**Group, 344

clipboard

- vector output support, 16

closing

- projects, 89

color coding, 115

command stream substitution, 117

command stream substitutions

- %, 30, 117

- for Macintosh, 30, 118

COMPILE_OPT statement, 153

compiler options, 23

compiling

- a file from a project, 93

- all files in a project, 101

- modified files in a project, 101

complex breakpoints, 110

compression for files, 24

ComputeGeometry method

- IDLanROI**, 289

ComputeMask method

- IDLanROI**, 291

- IDLanROI**Group, 313

ComputeMesh method

- IDLanROI**Group, 316

ContainsPoints method

- IDLanROI**, 294

- IDLanROI**Group, 311

creating

- .sav file from a project, 102

- IDL** Runtime distribution, 106

- projects, 87

CW, 156, 162, 164, 166, 172

CW_FILESEL function, 156

CW_LIGHT_EDITOR function, 158

CW_LIGHT_EDITOR_GET procedure, 162

CW_LIGHT_EDITOR_SET procedure, 164

CW_PALETTE_EDITOR function, 166

CW_PALETTE_EDITOR_GET procedure, 172

CW_PALETTE_EDITOR_SET procedure, 173

D

deleting

- files in a project, 92

developer's kit license, 107

DIALOG, 174, 176

DIALOG_READ_IMAGE function, 174

DIALOG_WRITE_IMAGE function, 176

distribution

- creating, 106

DLM, 178

DLM_LOAD procedure, 178

DRAW, 179

E

editing

a source file from a project, 93

editor

color/font coding, 115

ENABLE, 181

ENABLE_SYSRTN procedure, 181

endian

big, 132

byte ordering, 132

little, 132

EOS, 183, 185, 186, 187

EOS_GD_QUERY function, 183

EOS_PT_QUERY function, 185

EOS_QUERY function, 186

EOS_SW_QUERY function, 187

EXECUTE, 182

exporting

projects, 105

F

file

adding to a project, 90

compiling from a project, 93

compiling in a project, 101

compression, 24

editing from a project, 93

moving in a project, 91

removing from a project, 92

setting properties for a project, 94

file I/O improvements, 24

font coding, 115

G

GET, 24, 189

GET_DRIVE_LIST function, 189

GetDeviceInfo method

IDLgrBuffer, 328

IDLgrClipboard, 330

IDLgrVRML, 347

IDLgrWindow, 349

GetProperty method

IDLanROI, 296

IDLanROIGroup, 318

IDLgrROI, 335

GRID, 190

GRID_TPS function, 190

group

moving files in a project, 91

GZIP compression, 24

H

HDF files

importing using macros, 137

HDF improvements, 31

HDF SD compression, 31

HDF version, 32

HDF-EOS files

importing using macros, 137

HDF-EOS improvements, 31

HDF-EOS version, 32

HELP enhancements, 23

I

IDL, resetting the session, 23

IDLanROI

AppendData method, 286

Cleanup method, 288

ComputeGeometry method, 289

ComputeMask method, 291

ContainsPoints method, 294

GetProperty method, 296

Init method, 298

- RemoveData method, [300](#)
- ReplaceData method, [301](#)
- Rotate method, [303](#)
- Scale method, [304](#)
- SetProperty method, [305](#)
- Translate method, [306](#)
- IDLanROI object class, [284](#)
- IDLanROIGroup
 - Add method, [309](#)
 - Cleanup method, [310](#)
 - ComputeMask method, [313](#)
 - ComputeMesh method, [316](#)
 - ContainsPoints method, [311](#)
 - GetProperty method, [318](#)
 - Init method, [320](#)
 - Rotate method, [321](#)
 - Scale method, [322](#)
 - Translate method, [323](#)
- IDLanROIGroup object class, [307](#)
- IDLffLanguageCat
 - IsValid method, [325](#)
 - SetCatalog method, [327](#)
- IDLffLanguageCat object, [324](#)
- IDLffLanguageCatalog
 - Query method, [326](#)
- IDLgrBuffer
 - GetDeviceInfo method, [328](#)
- IDLgrClipboard
 - GetDeviceInfo method, [330](#)
- IDLgrROI
 - Cleanup method, [334](#)
 - GetProperty method, [335](#)
 - Init method, [336](#)
 - PickVertex method, [340](#)
 - SetProperty method, [341](#)
- IDLgrROI object class, [332](#)
- IDLgrROIGroup
 - Add method, [343](#)
 - Cleanup method, [344](#)
 - Init method, [345](#)
 - PickRegion method, [346](#)
- IDLgrROIGroup object class, [342](#)
- IDLgrVRML
 - GetDeviceInfo method, [347](#)
- IDLgrWindow
 - GetDeviceInfo method, [349](#)
- IMAGE, [193](#), [193](#)
- image processing
 - improvements, [10](#)
 - morphological functions, [11](#)
 - ROI improvements, [11](#)
- IMAGE_STATISTICS procedure, [193](#)
- images
 - macros for importing, [121](#)
- import macro
 - ASCII files, [125](#)
 - binary files, [131](#)
 - image files, [121](#)
 - scientific data formats, [137](#)
- Init method
 - IDLanROI, [298](#)
 - IDLanROIGroup, [320](#)
 - IDLgrROI, [336](#)
 - IDLgrROIGroup, [345](#)
- ISOCONTOUR, [196](#)
- ISOCONTOUR procedure, [196](#)
- ISOSURFACE procedure, [199](#)
- IsValid method
 - IDLffLanguageCat, [325](#)

K

keywords, new and updated, [53](#)

L

libraries, updated versions, [32](#)

license

developer's kit, [107](#)

LOCALE_GET function, 201

M

macro

importing

ASCII data, 125

binary data, 131

HDF files, 137

HDF-EOS files, 137

image files, 121

NETCDF files, 137

macros support, 117

main menu bar enhancements, 110

MESH, 202, 204, 206, 207, 209, 210, 212, 214, 214, 216

MESH_CLIP function, 202

MESH_DECIMATE function, 204

MESH_ISSOLID function, 206

MESH_MERGE function, 207

MESH_NUMTRIANGLES function, 209

MESH_SMOOTH function, 210

MESH_SURFACEAREA function, 212

MESH_VALIDATE function, 214

MESH_VOLUME function, 216

MORPH, 217, 222, 224, 226, 228, 229

MORPH_CLOSE function, 217

MORPH_DISTANCE function, 219

MORPH_GRADIENT function, 222

MORPH_HITORMISS function, 224

MORPH_OPEN function, 226

MORPH_THIN function, 228

MORPH_TOPHAT function, 229

morphological functions, 11

moving

files in a project, 91

MSG_CAT_CLOSE procedure, 231

MSG_CAT_COMPILE procedure, 232

MSG_CAT_OPEN function, 234

N

NETCDF files

importing using macros, 137

O

object class

IDLAnROI, 284

IDLAnROIGroup, 307

IDLffLanguageCat, 324

IDLgrROI, 332

IDLgrROIGroup, 342

OpenGL

querying for information, 18

opening

projects, 89

options

setting for project, 96

P

PARTICLE, 236

PARTICLE_TRACE procedure, 236

PDF files, 46

PickRegion method

IDLgrROIGroup, 346

PickVertex method

IDLgrROI, 340

platforms supported, 82

printer

vector output support, 16

project

adding files, 90

building, 102

closing, 89

compiling a file, 93

compiling all files, 101

compiling modified files, 101

creating, 87

creating a .sav file, 102

- editing source files, 93
- exporting, 105
- moving files, 91
- opening, 89
- removing files, 92
- running an application, 104
- saving, 89
- setting build order, 99
- setting file properties, 94
- setting options, 96
- storing source files, 86
- testing a .prc file, 94

projects

- overview, 27, 84

Q

QUERY, 25, 239, 242

Query method

- IDLffLanguageCat, 326

QUERY_IMAGE function, 239

QUERY_WAV function, 242

querying

- for OpenGL information, 18

R

READ, 25, 25, 147, 243, 243, 245, 247

READ_IMAGE function, 245

READ_WAV function, 247

reading

- ASCII data, 125
- binary data, 131
- data using macros, 121, 125, 131
- HDF files, 137
- HDF-EOS files, 137
- image files, 121
- NETCDF files, 137
- scientific format data, 137

- recomendations
 - storing files in a project, 86
- region of interest
 - IDLanROI, 284
 - improvements, 11
- RemoveData method
 - IDLanROI, 300
- removing
 - files in a project, 92
- ReplaceData method
 - IDLanROI, 301
- resetting and IDL session, 23
- ROI
 - improvements, 11
- Rotate method
 - IDLanROI, 303
 - IDLanROIGroup, 321
- running
 - application from a project, 104

S

saving

- projects, 89

Scale method

- IDLanROI, 304
- IDLanROIGroup, 322

scientific data format

- importing using macros, 137

SetCatalog method

- IDLffLanguageCat, 327

SetProperty method

- IDLanROI, 305
- IDLgrROI, 341

setting

- options for a project, 96
- properties of a file in a project, 94

sparse matrix, 19

storing

- file in a project, 86

STRCMP, 248

STRCMP function, [248](#)
 STREAMLINE, [250](#)
 STREAMLINE procedure, [250](#)
 STREGEX, [252](#)
 STREGEX procedure, [252](#)
 string processing, [22](#)
 STRJOIN, [256](#)
 STRJOIN function, [256](#)
 STRMATCH function, [257](#)
 STRSPLIT, [260](#)
 STRSPLIT function, [260](#)
 STRUCT, [264](#)
 STRUCT_HIDE procedure, [264](#)
 sub-rectangles support, [18](#)
 substitution, command stream, [117](#)
 supported platforms, [82](#)

T

testing
 .prc file from a project, [94](#)
 TETRA, [266](#), [268](#), [269](#)
 TETRA_CLIP function, [266](#), [266](#)
 TETRA_SURFACE function, [268](#)
 TETRA_VOLUME function, [269](#)
 text
 orientation for axis label, [18](#)

Translate method
 IDLanROI, [306](#)
 IDLanROIGroup, [323](#)

V

VALUE, [271](#)
 VALUE_LOCATE function, [271](#)
 VData access, [31](#)
 vector output of Object Graphics, [16](#)
 VECTOR_FIELD procedure, [273](#)
 VGroup access, [31](#)

W

WATERSHED, [275](#)
 WATERSHED function, [275](#)
 WAV support, [26](#)
 WRITE, [25](#), [277](#), [278](#), [278](#)
 WRITE_IMAGE procedure, [277](#)
 WRITE_WAV function, [278](#)

X

XOBJVIEW procedure, [279](#)

