IDL

What's New in IDL 6.0

RSI
Research Systems Inc.

IDL Version 6.0
July, 2003 Edition

0703IDL60WN

Restricted Rights Notice

Limitation of Warranty

Permission to Reproduce this Manual

Acknowledgments

IDL® is a registered trademark and ION™, ION Script™, ION Java™, are trademarks 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 1988-2001 The Board of Trustees of the University of Illinois
All rights reserved.

NCSA HDF5 (Hierarchical Data Format 5) Software Library and Utilities
Copyright 1998, 1999, 2000, 2001, 2002 by the Board of Trustees of the University of Illinois. All rights reserved.

CDF Library
Copyright © 1999
National Space Science Data Center
NASA/Goddard Space Flight Center

NetCDF Library
Copyright © 1993-1996 University Corporation for Atmospheric Research/Unidata

HDF EOS Library
Copyright © 1996 Hughes and Applied Research Corporation

This software is based in part on the work of the Independent JPEG Group.

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

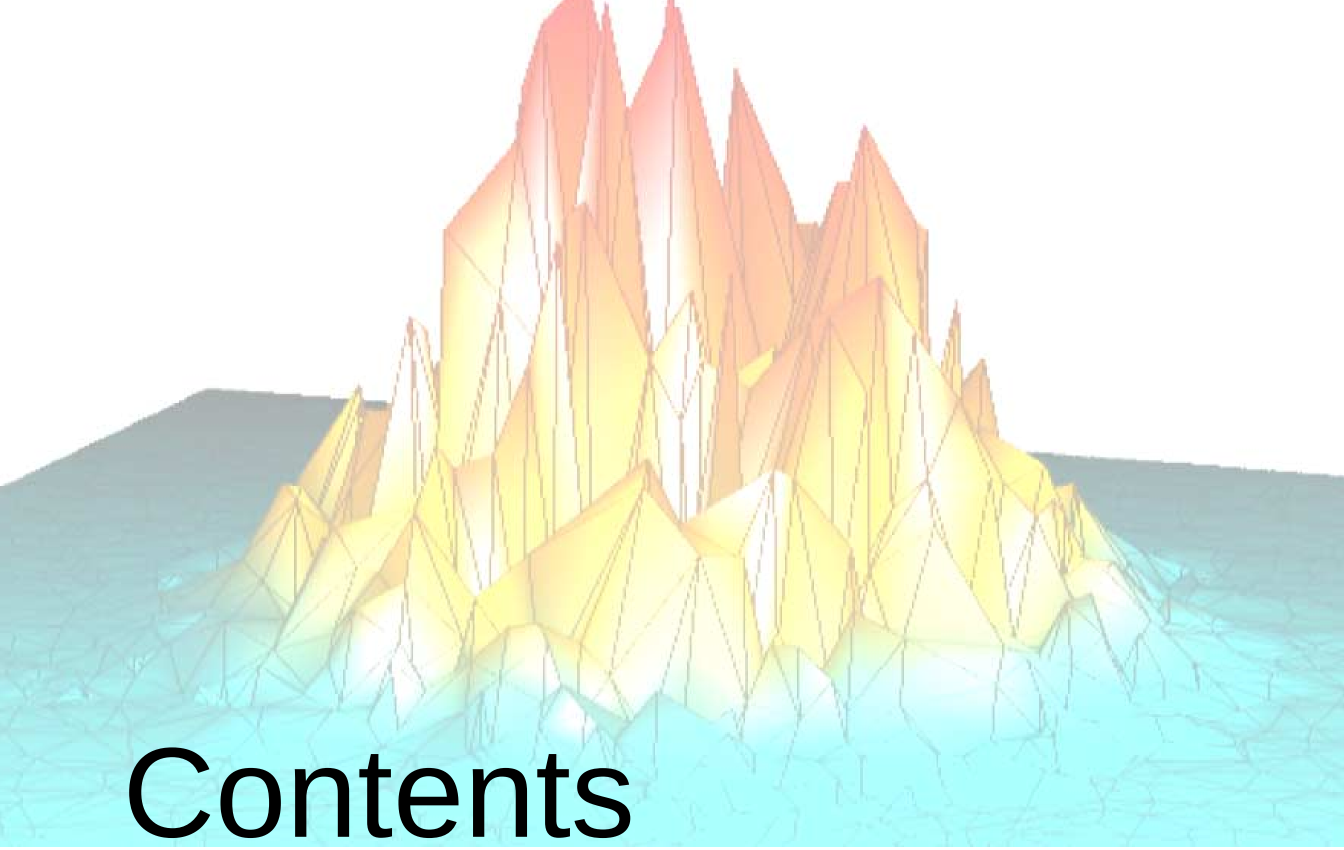
Use of this software for providing LZW capability for any purpose is not authorized unless user first enters into a license agreement with Unisys under U.S. Patent No. 4,558,302 and foreign counterparts. For information concerning licensing, please contact: Unisys Corporation, Welch Licensing Department - C1SW19, Township Line & Union Meeting Roads, P.O. Box 500, Blue Bell, PA 19424.

Portions of this computer program are copyright © 1995-1999 LizardTech, Inc. All rights reserved. MrSID is protected by U.S. Patent No. 5,710,835. Foreign Patents Pending.

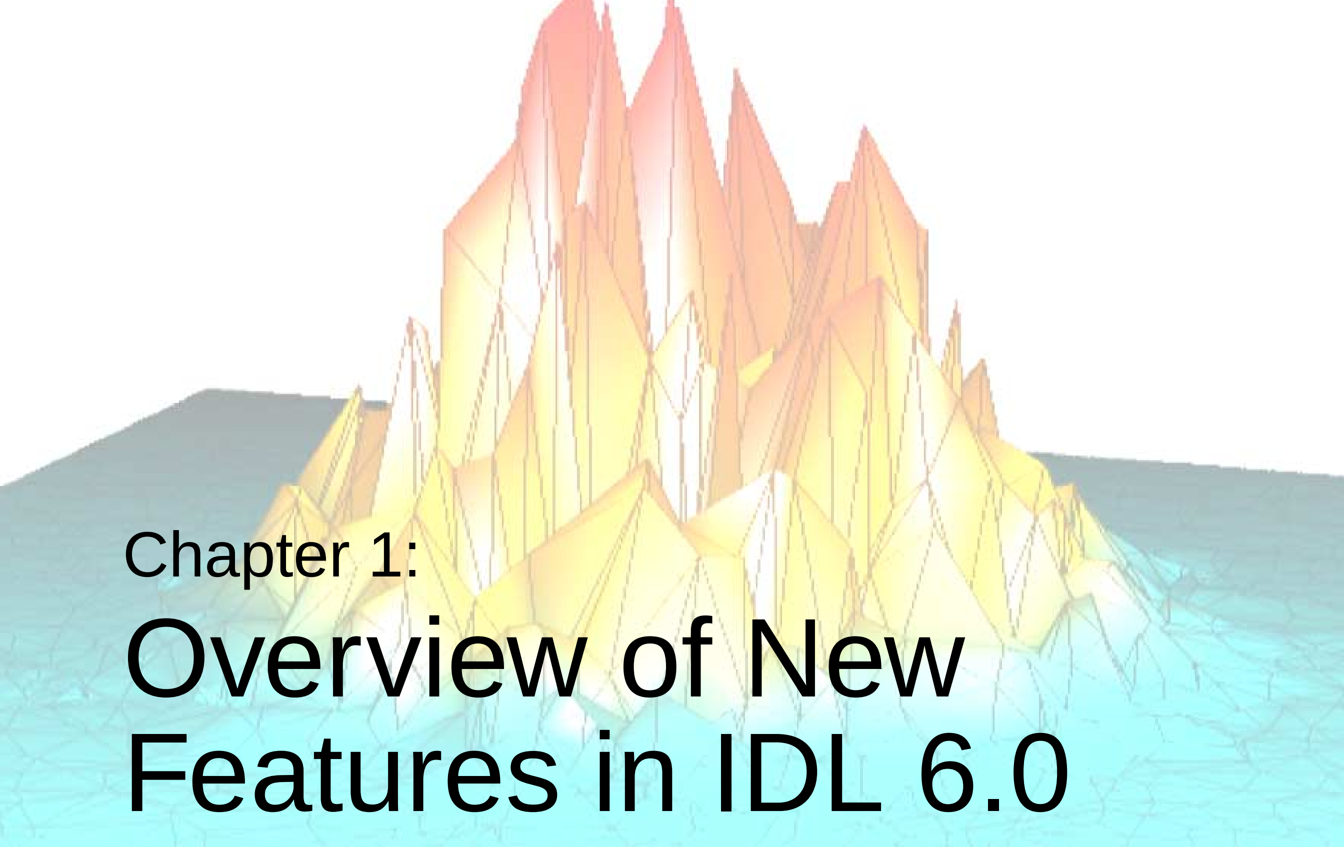
This product includes software developed by the Apache Software Foundation (http://www.apache.org/)

IDL Wavelet Toolkit Copyright © 2002 Christopher Torrence.

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



Contents



Chapter 1: Overview of New Features in IDL 6.0

This chapter contains the following topics:

New iTools for Interactive Analysis

Introducing the iTools

The new Intelligent Tools (iTools) are a set of interactive utilities that combine data analysis and visualization with the task of producing presentation quality graphics. Based on the IDL Object Graphics system, the iTools are designed to help you get the most out of your data with minimal effort. They allow you to continue to benefit from the control of a programming language, while enjoying the convenience of a point-and-click environment.

In IDL 6.0, five pre-built iTools are exposed for immediate interactive use. Each of these five tools is designed around a specific data or visualization type, including:

- Two and three dimensional plots (line, scatter, polar, and histogram style)
- Surface representations
- Contour lines
- Image displays
- Volume visualizations

The iTools are built upon a new object-oriented framework, or set of object classes, that serve as the building blocks for the interface and functionality of the Intelligent Tools. IDL programmers can easily use this framework to create custom data analysis and visualization environments. Such custom Intelligent Tools may be called from within a larger IDL application, or they may serve as the foundation for a complete application in themselves.

A Single Tool with Many Faces

What sets the Intelligent Tools apart from precursors such as the Live Tools (now obsolete with IDL 6.0) — and what gives them their optimal power, flexibility, and extensibility — is the cohesive, open architecture of the Intelligent Tools system. The iTools system is actually comprised of a single tool, which adapts to handle the data that you pass to it. The plot, surface, image, contour, and volume tools are simply shortcut configurations, which facilitate *ad hoc* data analysis and visualization. Each tool encapsulates the functionality (data operations, display manipulations, and visualization types) required to handle its data or visualization type. However, you are not constrained to work with a single data or visualization type. For example, using the Intelligent Tools system, you may start by bringing up a surface plot in a surface tool and then import scattered point data into the same plot to see the relationship between two datasets. Or, you may start with an image display, overlay contours from another dataset, and map both the image and contours onto a three-dimensional surface representation of a third dataset. By adding new data into an iTool, it is easy to end up with a hybrid tool that can handle complex, composite visualizations.

The main enhancements the new iTools provide are more mouse interactivity, WYSIWYG (What-You-See-Is-What-You-Get) printing, built-in analysis, undo-redo capabilities, layout control, and better-looking plots. These robust, pre-built tools reduce the amount of programming IDL users must do to create interactive visualizations. At the same time, the iTools integrate seamlessly with the IDL Command Line, user interface controls, and custom algorithms. In this way, the iTools maintain and enhance the control and flexibility IDL users rely on for data exploration, algorithm design, and rapid application development.

Foundation for the Future

As you will discover, the iTools are compelling new tools to add to your arsenal. They complement the strong foundation that IDL has maintained over the course of its evolution. This foundation has made possible countless valuable user-written applications across many disciplines and industries. However, the iTools also represent the start of a new, updated display paradigm for IDL. While the iTools system in IDL 6.0 is a powerful and flexible environment that will allow you to immediately accelerate your data interpretation and reporting, it is only the beginning. We will continue to build on this new technology in future releases. You can look forward to more functionality, flexibility, and optimization as the iTools system continues to grow.

We look forward to members of the IDL community building on the iTools system as well. The iTools source code is included in the IDL distribution to allow you to:

- extend the pre-built tools with your own operations, manipulations, visualization types, and GUI controls,
- create your own custom tools based on the iTools component framework,
- share your inventions with others in the IDL community via the RSI User-Contributed Library (http://www.RSInc.com/codebank) or other avenues of collaboration and distribution.

New iTool Routines

Five new iTool routines allow access to the pre-built Intelligent Tools from the IDL Command Line or within user-written code. The routines accept data parameters and keywords to control the initial characteristics and allow for overplotting. Data access and visualization properties can also be controlled interactively via the iTool user interface.

The following iTool routines are a part of IDL 6.0:

- IPLOT - for two and three-dimensional plotting of line and point data. For more details, see "IPLOT" in the *IDL Reference Guide* manual.
- ISURFACE - for surface representations of two-dimensional array data and irregularly sampled point collections. For more details, see "ISURFACE" in the *IDL Reference Guide* manual.
- ICONTOUR - for the production and manipulation of contour maps of two-dimensional array data and irregularly sampled point collections. For more details, see "ICONTOUR" in the *IDL Reference Guide* manual.
- IIMAGE - for image display, exploration, ROI definition, and basic processing. For more details, see "IIMAGE" in the *IDL Reference Guide* manual.
- IVOLUME - for volume rendering, manipulation, and dissection. For more details, see "IVOLUME" in the *IDL Reference Guide* manual.

New iTool Object Classes

The new iTool object classes allow programmers to leverage the underlying iTool component framework. Using these building blocks, you can create custom iTools from scratch or extend existing iTools with your own operations, manipulations, visualization types, and GUI controls.

The following iTool objects are a part of IDL 6.0. These object are described in the *IDL Reference Guide*:

- IDLitCommand
- IDLitCommandSet
- IDLitComponent
- IDLitContainer
- IDLitData

- IDLitDataContainer
- IDLitDataOperation
- IDLitIMessaging
- IDLitManipulator
- IDLitManipulatorContainer
- IDLitManipulatorManager
- IDLitManipulatorVisual
- IDLitOperation
- IDLitParameter
- IDLitParameterSet
- IDLitReader
- IDLitTool
- IDLitUI
- IDLitVisualization
- IDLitWindow
- IDLitWriter

ITools User’s and Developer’s Guides

With the introduction of the Intelligent Tools, IDL 6.0 includes two new manuals: The *iTools User’s Guide* walks you through calling the iTools and using the iTools system interactively. The *iTools Developer’s Guide* instructs you on how to use the iTools component framework to develop your own iTools or build on existing ones.

New IDL Virtual Machine

RSI now offers a freely distributable utility known as the *IDL Virtual Machine*. The IDL Virtual Machine is designed to provide IDL users with a simple, no-cost method for distributing IDL applications to colleagues and customers. It runs on all IDL supported platforms (see "Requirements for this Release" on page 117) and does not require a license to run.

The IDL Virtual Machine will run a compiled IDL `.sav` file even if no IDL license is present. RSI's aim with the IDL Virtual Machine is to facilitate IDL code collaboration and application distribution. However, a few restrictions exist:

- The IDL Virtual Machine displays a splash screen on startup.
- `.sav` files must be created using IDL version 6.0 or later.
- No access to the IDL command line or IDL compiler is provided.
- IDL programs that call the EXECUTE function will not run in the IDL Virtual Machine.
- Callable IDL applications and applications that use the IDL ActiveX control will not run in the IDL Virtual Machine.
- The IDL Virtual Machine must be installed via the installation program provided by RSI. You are prohibited from modifying the IDL Virtual Machine distribution.

See the *Building IDL Applications* manual for more information on creating applications for the IDL Virtual Machine.

Getting the IDL Virtual Machine

The IDL Virtual Machine is included with all IDL distributions, including the freely-downloadable IDL installer available from RSI's website (http://www.RSInc.com). During installation, you can choose to install either a full IDL distribution (which includes the IDL Virtual Machine) or just the IDL Virtual Machine distribution.

Using the IDL Virtual Machine

When you attempt to run a `.sav` file, IDL will first attempt to execute the file using a licensed version of IDL. If no licenses are available, the `.sav` file will be executed in IDL Virtual Machine mode.

To explicitly use the IDL Virtual Machine when an IDL license is present (for debugging purposes for example), do one of the following:

Windows Platforms

- Drag the `.sav` file to the IDL Virtual Machine desktop icon.
- At the command prompt, use the following command:

  ```
  idlrt -vm=file.sav
  ```

  where *file.sav* is the name of the IDL `.sav` file.

UNIX Platforms

- At the command prompt, use the following command:

  ```
  idl -vm=file.sav
  ```

  where *file.sav* is the name of the IDL `.sav` file.

New VM Keyword to the LMGR Routine

The LMGR function has a new VM keyword that allows you to test whether the current IDL session is running in IDL Virtual Machine mode. IDL Virtual Machine applications do not provide access to the IDL Command Line.

See "LMGR" on page 106 for more details.

New IDL-Java Bridge

IDL now supports the use of Java objects. You can access Java objects within your IDL code using the IDL-Java bridge, a built-in feature of IDL 6.0. The IDL-Java bridge enables you to take advantage of functionality provided by Java, including Java I/O, networking, and third party functionality.

The new IDLjavaObject class instantiates a desired Java object using the object's class name. An instance of this object within IDL allows you access methods and data members (properties) of the desired Java object. The IDLjavaObject class is defined in "IDLjavaObject" in the *IDL Reference Guide* manual.

To the IDL user, an instance of the IDLjavaObject class behaves just like any other IDL object. You can read about the creation and management of Java objects within IDL in Chapter 8, "Using Java Objects in IDL" in the *External Development Guide* manual.

When an instance of the IDLjavaObject class is created, the IDL-Java bridge connects that instance to a Java object. This initial connection starts a Java session. In IDL, you can monitor the session through the IDLJavaBridgeSession object. This object can be used to handle any exceptions (caused by the Java object) within IDL. This bridge session object is also described in Chapter 8, "Using Java Objects in IDL" in the *External Development Guide* manual.

Currently, the IDL-Java bridge is supported on the Windows, Linux, Solaris, and Macintosh platforms supported in IDL. See "Requirements for this Release" on page 117 for more information on these platforms supported in IDL 6.0.

New Path Caching

The first time an IDL session attempts to call a function or procedure written in the IDL language, it must locate the file containing the code for that routine and compile it. The file containing the routine must have the same name as the routine, with either a .pro or a .sav extension. After trying to open the file in the user's current working directory, IDL will attempt to open the file in each of the directories given by the !PATH system variable, in the order specified by !PATH. The search stops with the first directory containing a file with the desired name.

IDL now maintains an in-memory cache of the .pro and .sav files located in directories referenced via the !PATH system variable. This path cache is built automatically during normal operation, as IDL searches the directories specified by !PATH to locate the code for IDL routines required at runtime. The path cache operates on a per-directory basis. The current contents of the path cache can be viewed using the HELP, /PATH_CACHE command. See "Enhancement to the HELP Routine" on page 35 for more details.

Once a directory is cached, IDL knows whether or not it contains a given file, without the need to actually attempt to open that file. This information allows IDL to skip the open attempt in directories that do not have the desired file. As such, the path cache can provide a significant boost in the speed of path searching. The startup speed of large object oriented applications, is significantly improved by the path cache, as method resolution requires heavy path searching activity.

The PATH_CACHE procedure is used to control IDL's use of the path cache. In almost all cases, the operation of the path cache is transparent to the IDL user, save for the boost in path searching speed it provides. The cache automatically adjusts to changes made to the setting of !PATH without the need for manual intervention. Hence, PATH_CACHE should not be necessary in typical IDL operation. It exists to allow complete control over the details of how and when the caching operation is performed. See "PATH_CACHE" in the *IDL Reference Guide* manual for more details.

Visualization Enhancements

The following enhancements have been made to IDL's visualization functionality for the 6.0 release:

- Object Graphics Font Rendering Improvements
- New Depth Buffer Controls for Graphic Objects

Object Graphics Font Rendering Improvements

IDL 6.0 incorporates the FreeType Library for improved rendering of Object Graphics fonts. Previously, characters in an IDLgrText object were rendered by tessellating each glyph outline into a set of small triangles. IDL 6.0 renders an entire IDLgrText string as a high quality bitmap, which is texture mapped onto a single polygon. This technique allows for clearer characters at any size, easier manipulations, background colors, kerning, and blending. For information on the FreeType Project, visit http://www.freetype.org.

The properties to IDLgrText related to this new font rendering are:

- ALPHA_CHANNEL
- FILL_BACKGROUND
- FILL_COLOR
- KERNING
- RENDER_METHOD

Note

FreeType font rendering is now the default text rendering method in IDL. If you need to switch back to the triangle method, the RENDER_METHOD property can be used to change the type of font rendering.

For more details on these properties, see "IDLgrText" on page 81.

Examples: Font Rendering Improvements

The following example routines show how to use the new font properties to the IDLgrText object. While running these routines, you can proceed to the next display by quitting (**File** → **Quit**) out of the current XOBJVIEW display.

This first example compares simple font rendering tasks in IDL 6.0 and previous versions of IDL.

```
PRO ExCompareSimpleFonts

; Create previous version text object and IDL 6.0 text
; object (with kerning applied).
oText1 = OBJ_NEW('IDLgrText', 'IDL 5.6', $
   RENDER_METHOD = 1, LOCATIONS = [0, 0.05, 0])
oText2 = OBJ_NEW('IDLgrText', 'IDL 6.0', $
   LOCATIONS = [0, -0.05, 0], /KERNING)

; Show the text objects.
oModel = OBJ_NEW('IDLgrModel')
oModel -> Add, oText1
oModel -> Add, oText2
XOBJVIEW, oModel, /BLOCK

; Put a polygon behind the text and give the IDL 6.0 text
; a background color.
oText2 -> SetProperty, FILL_BACKGROUND = 1, $
   FILL_COLOR = [0, 255, 0]
oPoly = OBJ_NEW('IDLgrPolygon', [-0.1, 0.1, 0.1, -0.1], $
   [0.1, 0.1, -0.1, -0.1], [0., 0., 0., 0.], $
   COLOR = [255, 0, 0], /DEPTH_OFFSET)

; Show the polygon and updated text objects.
oModel -> Add, oPoly
XOBJVIEW, oModel, /BLOCK

; Make the text semi-transparent to let the polygon
; show through and display the results.
oText2 -> SetProperty, ALPHA_CHANNEL = 0.5
XOBJVIEW, oModel, /BLOCK

; Cleanup.
OBJ_DESTROY, [oModel]

END
```

The following figure shows the results of this example:

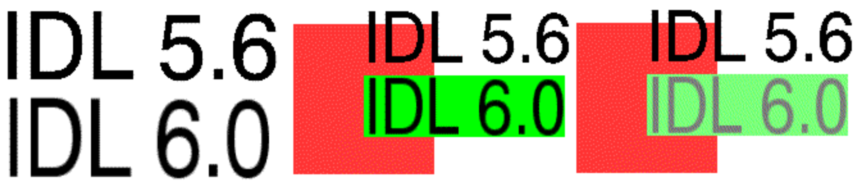


Figure 1-1: FreeType Rendering Compared to Previous Rendering

The following example is more involved than the previous one. It shows how to use IDL 6.0's font rendering improvements to clarify cluttered axis and data labels.

```
PRO ExFreeTypeAxes

; Create data.
angle = 2.*!PI*(0.5 - (FINDGEN(37)/36.))
amplitude = 5*SIN(angle)

; Create a model to contain the plots, axes, labels, and titles
; to that model.
oModel = OBJ_NEW('IDLgrModel')

; Create plots and add them to the model.
oPlots = OBJARR(5)
FOR i = 0, 4 DO BEGIN
   oPlots[i] = OBJ_NEW('IDLgrPlot', angle, $
      (amplitude*(1. - (i*0.05))))
ENDFOR
oPlots[0] -> GetProperty, XRANGE = xRange, YRANGE = yRange
oModel -> Add, oPlots

; Create axes and add them to the model.
oXAxis = OBJ_NEW('IDLgrAxis', 0, $
   RANGE = xRange, /EXACT)
oModel -> Add, oXAxis
oYAxis = OBJ_NEW('IDLgrAxis', 1, $
   RANGE = yRange, /EXACT)
oModel -> Add, oYAxis

; Create labels and add them to the model.
xLabel = 2.*!PI*(0.5 - (FINDGEN(20)/19.))
yLabel = 5*SIN(xLabel)
oLabels = OBJARR(N_ELEMENTS(xLabel))
FOR i = 0, (N_ELEMENTS(oLabels) - 1) DO BEGIN
```

```
   oLabels[i] = OBJ_NEW('IDLgrText', $
      STRTRIM(xLabel[i],2), $
      LOCATION = [xLabel[i], yLabel[i]], $
      CHAR_DIMENSIONS = [0.3, 0.3], $
      ALIGNMENT = 0.5)
ENDFOR
oModel -> Add, oLabels

; Create titles and add them to the model.
oXTitle = OBJ_NEW('IDLgrText', $
   'Alpha = Angle (Radians)', LOCATION = [0, yRange[0]], $
   ALIGNMENT = 0.5, CHAR_DIMENSIONS = [0.5, 0.5], $
   ZCOORD_CONV = [0.1, 1])
oModel -> Add, oXTitle
oYTitle = OBJ_NEW('IDLgrText', $
   'Epsilon = Amplitude (Centimeters)', $
   LOCATION = [xRange[0], 0], ALIGNMENT = 0.5,  $
   CHAR_DIMENSIONS = [0.5,0.5], BASELINE = [0, 1], $
   UPDIR = [-1, 0], ZCOORD_CONV = [0.1, 1])
oModel -> Add, oYTitle
oTitle = OBJ_NEW('IDLgrText', 'SINE WAVE', $
   ALIGNMENT = 0.5, VERTICAL_ALIGNMENT  = -1., $
   LOCATIONS = [MEAN(xRange), yRange[1]], $
   CHAR_DIMENSIONS = [0.7, 0.7])
oModel -> Add, oTitle

; Display the model.
XOBJVIEW, oModel, /BLOCK, SCALE = 0.9, $
   TITLE = 'Original Display'

; Make the axis titles translucent.
oXTitle -> SetProperty, /FILL_BACKGROUND, $
   FILL_COLOR = [230, 230, 230], ALPHA_CHANNEL = 0.5
oYTitle -> SetProperty, /FILL_BACKGROUND, $
   FILL_COLOR = [230, 230, 230], ALPHA_CHANNEL = 0.5

; Make the labels translucent.
FOR i = 0, (N_ELEMENTS(oLabels) - 1) DO BEGIN
   oLabels[i] -> SetProperty, ALPHA_CHANNEL = 0.6
ENDFOR

; Display the modified model.
XOBJVIEW, oModel, /BLOCK, SCALE = 0.9, $
   TITLE = 'Improved Display'

; Cleanup object references.
OBJ_DESTROY, oModel

END
```

New Depth Buffer Controls for Graphic Objects

In graphics rendering, the depth buffer is an array of depth values maintained by a graphics device, one value per pixel, to record the depth of primitives rendered at each pixel. It is usually used to prevent the drawing of objects located behind other objects that have already been drawn in order to generate a visually correct scene. In IDL, smaller depth values are closer to the viewer.

New properties to graphic objects in IDL 6.0 provide more control over how Object Graphics primitives are affected by the depth buffer. You can now control which primitives may be rejected from rendering by the depth buffer, how the primitives are rejected, and which primitives may update the depth buffer.

Control of the depth buffer is achieved through a test function or by completely disabling the buffer. The depth test function is a logical comparison function used by the graphics device to determine if a pixel should be drawn on the screen. This decision is based on the depth value currently stored in the depth buffer and the depth of the primitive at that pixel location.

The test function is applied to each pixel of an object. A pixel of the object is drawn if the object's depth at that pixel passes the test function set for that object. If the pixel passes the depth test, the depth buffer value for that pixel is also updated to the pixel's depth value.

The possible test functions are:

- INHERIT - use the test function set for the parent model or view.
- NEVER - never passes.
- LESS - passes if the depth of the object's pixel is less than the depth buffer's value.
- EQUAL - passes if the depth of the object's pixel is equal to the depth buffer's value.
- LESS OR EQUAL - passes if the depth of the object's pixel is less than or equal to the depth buffer's value.
- GREATER - passes if the depth of the object's pixel is greater than or equal to the depth buffer's value.
- NOT EQUAL - passes if the depth of the object's pixel is not equal to the depth buffer's value.

- GREATER OR EQUAL - passes if the depth of the object’s pixel is greater than or equal to the depth buffer’s value.
- ALWAYS - always passes

The IDL default is LESS. Commonly used values are LESS and LESS OR EQUAL, which allow primitives closer to the viewer to be drawn.

Disabling the depth test function allows all primitives to be drawn on the screen without testing their depth against the values in the depth buffer. When the depth test is disabled, the graphics device effectively uses the painter’s algorithm to update the screen. That is, the last item drawn at a location is the item that remains visible. The depth test function of ALWAYS produces the same result as disabling the depth test.

Moreover, you can disable updating the depth buffer. Disabling depth buffer writing prevents the updating of depth information as primitives are drawn to the frame buffer. Such primitives are unprotected in the sense that any other primitive drawn later at that location will draw over it as if it were not there.

Most atomic graphics objects now have the following new properties related to the depth buffer:

- DEPTH_TEST_DISABLE
- DEPTH_TEST_FUNCTION
- DEPTH_WRITE_DISABLE

For more details on these properties, see “New IDL Object Properties” on page 45.

Analysis Enhancements

The following enhancements have been made to IDL’s data analysis functionality for the 6.0 release:

- New FITA and STATUS Keywords to CURVEFIT
- New MEASURE_ERRORS Keyword to GAUSSFIT
- New PIXEL_CENTER Keyword for ROI Masks
- Enhancements to the INTERVAL_VOLUME, ISOSURFACE, and MESH_DECIMATE Routines

New FITA and STATUS Keywords to CURVEFIT

The FITA keyword to the CURVEFIT routine allows you to specify parameters that should remain fixed.

The STATUS keyword to the CURVEFIT routine specifies a named variable that will contain an integer indicating the status of the computation.

See “CURVEFIT” on page 99 for more details.

New MEASURE_ERRORS Keyword to GAUSSFIT

A new MEASURE_ERRORS keyword has been added to GAUSSFIT, which allows you to pass in a vector of standard measurement errors for each data point. Prior to IDL6.0, GAUSSFIT would assume measurement errors of 1.0 for each point. Now, if MEASURE_ERRORS is not specified, the measurement errors are assumed to be zero.

See “GAUSSFIT” on page 101 for more details.

New PIXEL_CENTER Keyword for ROI Masks

You can now fine tune the offset between pixels and coordinates for ROI vertices. The PIXEL_CENTER allows you to specify the location of the lower-left mask pixel.

For more details, see “IDLanROI::ComputeMask” on page 96.

Enhancements to the INTERVAL_VOLUME, ISOSURFACE, and MESH_DECIMATE Routines

The INTERVAL_VOLUME procedure, ISOSURFACE procedure, and MESH_DECIMATE function now provide a way to monitor the progress of the algorithms performed by these routines; using the PROGRESS_CALLBACK, PROGRESS_METHOD, PROGRESS_OBJECT, PROGRESS_PERCENT, and PROGRESS_USERDATA keywords. By using these keywords, you can present progress bars to your users during the execution of these routines. See "INTERVAL_VOLUME" on page 102, "ISOSURFACE" on page 104, and "MESH_DECIMATE" on page 107 for details.

Language Enhancements

The following enhancements have been made to the core of the IDL Language for the 6.0 release:

- Increment and Decrement Operators
- Compound Assignment Operators
- New Logical Operators
- New Logical Operation Functions
- LOGICAL_PREDICATE Compilation Option
- Multiple Subscripts Now Allowed On Assignment ASSOC Variables
- IEEE Floating Point NaN Comparisons Give Correct Results Under Microsoft Windows
- Enhancement to the ARRAY_EQUAL Routine
- Enhancement to the HELP Routine
- Enhancement to the MESSAGE Routine
- Enhancement to the RESOLVE_ALL Routine
- Enhancement to the SHMMAP Routine
- Enhancement to the STRSPLIT Routine
- New ARRAY_INDICES Function
- New IDL_VALIDNAME Function

Increment and Decrement Operators

IDL now includes increment (++) and decrement (--) operators that can be applied to variables of any numeric type. The ++ operator increments the target variable by one. The -- operator decrements the target by one.

Increment and decrement operators can be used, along with a variable, as standalone statements:

- `A++` or `++A`
- `A--` or `--A`

The increment or decrement operator may be placed either before or after the target variable. The same operation is carried out in either case. These operators are very efficient, since the variable is incremented *in place* and no temporary copies of the data are made.

The increment and decrement operators can also be used within expressions. When the operator follows the target expression, it is applied *after* the value of the target is evaluated for use in the surrounding expression. When the operator precedes the target expression, it is applied *before* the value of the target is evaluated for use in the surrounding expression. For example, after executing the following statements, the value of the variable A is 27, while B is 28:

```
B = 27
A = B++
```

In contrast, after executing the following statements, both A and B have a value of 26:

```
B = 27
A = --B
```

Although the standalone statement and expression forms are very similar, the expression form has some efficiency and side-effect issues that do not apply to the statement form. See "Increment/Decrement" in Chapter 2 of the *Building IDL Applications* manual for details.

Compound Assignment Operators

IDL now supports the following compound assignment operators:

##=	#=	*=	+=	-=
/=	<=	>=	AND=	EQ=
GE=	GT=	LE=	LT=	MOD=
NE=	OR=	XOR=	^=	

These compound operators combine assignment with another operator. A statement such as:

```
A op= expression
```

where *op* is an IDL operator that can be combined with the assignment operator to form one of the above-listed compound operators, and *expression* is any IDL expression, produces the same result as the statement:

```
A = A op (expression)
```

The statement using the compound operator makes more efficient use of memory because it performs the operation on the target variable A *in place*. In contrast, the statement using the simple operators makes a copy of the variable A, performs the operation on the copy, and then assigns the resulting value back to A, temporarily using extra memory.

The following statement:

```
A op= expression
```

is equivalent to the IDL statement:

```
A = TEMPORARY(A) op (expression)
```

which uses the TEMPORARY function to avoid making a copy of the variable A. While there is no efficiency benefit to using the compound operator rather than the TEMPORARY function, the compound operator allows you to write the same statement more succinctly.

Compound Operators and Whitespace

When using the compound operators that include an operator referenced by a *keyword* rather than a *symbol* (AND=, for example), you must be careful to use whitespace between the operator and the target variable. Without appropriate whitespace, the result will not be what you expect. Consider the difference between these two statements:

```
AAND= 23
A AND= 23
```

The first statement assigns the value 23 to a variable named AAND. The second statement performs the AND operation between A and 23, storing the result back into A.

Compound operators that do not involve IDL keywords (+=, for example) do not require whitespace in order to be properly parsed by IDL, although such whitespace is recommended for code readability. That is, the statements

```
A+=23
A += 23
```

are identical, but the latter is more readable.

New Logical Operators

There are three new logical operators in IDL: &&, ||, and ~.

&&

The logical `&&` operator performs the logical short-circuiting "and" operation on two scalars or one-element arrays, returning 1 if both operands are true and 0 if either operand is false.

||

The logical `||` operator performs the logical short-circuiting "or" operation on two scalars or one-element arrays, returning 1 if either of the operands is true and 0 if both are false.

~

The logical ~ operator performs the logical "not" operation on a scalar or array operand. If the operand is a scalar, it returns scalar 1 if the operand is false or scalar 0 if the operand is true. If the operand is an array, it returns an array containing a 1 for each element of the operand array that is false, and a 0 for each element that is true.

Note

Programmers familiar with the C programming language, and the many languages that share its syntax, may expect ~ to perform bitwise negation (1's complement), and for `!` to be used for logical negation. This is not the case in IDL: `!` is used to reference system variables, the NOT operator performs bitwise negation, and ~ performs logical negation.

When is an Operand True?

When evaluated by a logical operator, an expression is considered to be "true" under the following conditions:

- For numerical operands, if the value is non-zero.
- For string operands, if the value is non-null.
- For heap variables (pointers and object references), if the point or object reference is non-null.

Short-circuiting

The `&&` and `||` logical operators are *short-circuiting* operators. This means that IDL does not evaluate the second operand unless it is necessary in order to determine the proper overall answer. Short-circuiting behavior can be powerful, since it allows you to base the decision to compute the value of the second operand on the value of the first operand. For instance, in the expression:

```
Result = Op1 && Op2
```

IDL does not evaluate `Op2` if `Op1` is false, because it already knows that the result of the entire operation will be false. Similarly in the expression:

```
Result = Op1 || Op2
```

IDL does not evaluate `Op2` if `Op1` is true, because it already knows that the result of the entire operation will be true.

If you want to ensure that both operands are evaluated (perhaps because the operand is an expression that changes value when evaluated), use the new LOGICAL_AND and LOGICAL_OR functions (described in the next section) or the bitwise AND and OR operators.

New Logical Operation Functions

IDL 6.0 introduces three new functions that perform logical Boolean operations on their arguments:

LOGICAL_AND

The new LOGICAL_AND function performs a logical AND operation on its arguments. It returns True (1) if both of its arguments are non-zero (non-NULL for strings and heap variables), or False (0) otherwise.

Unlike the `&&` operator, LOGICAL_AND does not *short-circuit* when evaluating its arguments. Both arguments are always evaluated.

Note

LOGICAL_AND always returns either 0 or 1, unlike the AND operator, which performs a bitwise AND operation on integers, and returns one of the two arguments for other types.

For more information, see “LOGICAL_AND” in the *IDL Reference Guide* manual.

LOGICAL_OR

The new LOGICAL_OR function performs a logical OR operation on its arguments. It returns True (1) if either of its arguments are non-zero (non-NULL for strings and heap variables), and False (0) otherwise.

Unlike the `||` operator, LOGICAL_OR does not *short-circuit* when evaluating its arguments. Both arguments are always evaluated.

Note

LOGICAL_OR always returns either 0 or 1, unlike the OR operator, which performs a bitwise OR operation on integers, and returns one of the two arguments for other types.

For more information, see "LOGICAL_OR" in the *IDL Reference Guide* manual.

LOGICAL_TRUE

The new LOGICAL_TRUE function returns True (1) if its arguments are non-zero (non-NULL for strings and heap variables), and False (0) otherwise.

For more information, see "LOGICAL_TRUE" in the *IDL Reference Guide* manual.

LOGICAL_PREDICATE Compilation Option

The COMPILE_OPT statement allows you 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. The LOGICAL_PREDICATE compilation option has been added in IDL 6.0.

When running a routine compiled with the LOGICAL_PREDICATE option set, from the point where the COMPILE_OPT statement appears until the end of the routine, IDL will treat any non-zero or non-NULL predicate value as "true," and any zero or NULL predicate value as "false."

Background

A predicate expression is an expression that is evaluated as being “true” or “false” as part of a statement that controls program execution. IDL evaluates such expressions in the following contexts:

- `IF...THEN...ELSE` statements
- `? :` inline conditional expressions
- `WHILE...DO` statements
- `REPEAT...UNTIL` statements
- when evaluating the result from an INIT function method to determine if a call to OBJ_NEW successfully created a new object

By default, IDL uses the following rules to determine whether an expression is true or false:

- **Integer** — An integer is considered true if its least significant bit is 1, and false otherwise. Hence, odd integers are true and even integers (including zero) are false. This interpretation of integer truth values is sometimes referred to as “bitwise,” reflecting the fact that the value of the least significant bit determines the result.
- **Other** — Non-integer numeric types are true if they are non-zero, and false otherwise. String and heap variables (pointers and object references) are true if they are non-NULL, and false otherwise.

The LOGICAL_PREDICATE option alters the way IDL evaluates predicate expressions. When LOGICAL_PREDICATE is set for a routine, IDL uses the following rules to determine whether an expression is true or false:

- **Numeric Types** — A number is considered true if its value is non-zero, and false otherwise.
- **Other Types** — Strings and heap variables (pointers and object references) are considered true if they are non-NULL, or false otherwise.

Note on the NOT Operator

When using the LOGICAL_PREDICATE compile option, you must be aware of the fact that applying the IDL NOT operator to integer data computes a *bitwise* negation (1’s complement), and is generally not applicable for use in logical computations. Consider the common construction:

```
WHILE (NOT EOF(lun)) DO BEGIN
...
ENDWHILE
```

The EOF function returns 0 while the file specified by LUN has data left, and returns 1 when hits the end of file. However, the expression “NOT 1” has the numeric value -2. When the LOGICAL_PREDICATE option is not in use, the WHILE statement sees -2 as false; if the LOGICAL_PREDICATE is in use, -2 is a true value and the above loop will not terminate as desired.

The proper way to write the above loop uses the ~ logical negation operator:

```
WHILE (~EOF(lun)) DO BEGIN
...
ENDWHILE
```

It is worth noting that this version will work properly whether or not the LOGICAL_PREDICATE compile option is in use. Logical negation operations should always use the ~ operator in preference to the NOT operator, reserving NOT exclusively for bitwise computations.

Multiple Subscripts Now Allowed On Assignment ASSOC Variables

An associated variable (created via the ASSOC function) is a variable that maps the structure of an IDL array or structure variable onto the contents of a file. The file is treated as an array of these repeating units of data. The first array or structure in the file has an index of 0, the second as an index of 1, and so on. Such variables do not keep data in memory like a normal variable. Instead, when an associated variable is subscripted with the index of the desired array or structure within the file, IDL performs the input/output operation required to access the data. In all cases, the entire array associated with an index is input or output as a complete unit.

Previous versions of IDL allow you to specify additional subscripts in addition to the array index when fetching data in order to extract sub-elements of the array. This is implemented by reading the entire array into memory, and then performing the subscripted fetch operation on this in memory copy. This ability was allowed on data input only - only a single array index was allowed when writing data back. Hence, data could be written to an ASSOC variable only as a complete array. This limitation has been removed with IDL 6.0. Multiple subscripts can now be specified both reading and writing.

The following statements use an associated variable of 10x10 arrays in the file `data.dat` to illustrate:

```
OPENW, unit, 'data.dat', /GET_LUN  ; Open file.
A = ASSOC(unit, FLTARR(10, 10))    ; Associate variable.
A[1] = FINDGEN(10)                 ; Write findgen array value
                                   ; to file at index 1.
B = A[2, 3, 1]                     ; Read data element [2,3]
                                   ; from the array at index 1.
A[2, 3, 1] = 1001.7                ; Write new value to data
                                   ; element [2,3] of array at
                                   ; index 1.
```

The final statement above is allowed by IDL 6.0, but not by previous versions. It is implemented by reading the entire array at the specified index into memory, performing the subscripted store operation on the in-memory copy, and then writing the entire array back to the file at the specified index.

Note

Although notationally convenient, specifying multiple subscripts to ASSOC variables can be inefficient due to all the implicit Input/Output it generates. For large numbers of such accesses, we recommend reading the entire array into memory once, performing all the operations on the in-memory variable, and then writing the array back.

IEEE Floating Point NaN Comparisons Give Correct Results Under Microsoft Windows

According to the IEEE floating point standard, the Not A Number value (NaN) has the unique property that it is not equal to any other number, including itself. In IDL terms, this means that the expression:

```
!VALUES.F_NAN EQ !VALUES.F_NAN
```

should yield the value False (0). Making this work is the shared responsibility of the underlying hardware, operating system, and language compiler used to build a program (in the case of IDL, of the C/C++ compiler used to build IDL). Many programs use this identity to locate the NaN values within data.

Previous Microsoft C/C++ compilers did not generate floating point code that works properly in this case, and comparisons of NaN with itself would incorrectly yield True (1). IDL 6.0 is built with the latest Microsoft Visual C/C++ 7.0 compiler, which generates correct floating point code for IEEE comparisons. Hence, IDL 6.0 for Microsoft Windows correctly evaluates NaN comparisons.

This means that IDL 6.0 properly evaluates NaN comparisons on all supported platforms, the first release in IDL history for which this has been true.

Enhancement to the ARRAY_EQUAL Routine

The ARRAY_EQUAL function now also works on pointer and object references. When ARRAY_EQUAL is used with these types of references, it compares the *references*, not the heap variables to which the references point.

Enhancement to the HELP Routine

The HELP procedure has been enhanced with the PATH_CACHE keyword, which allows you to display a list of directories currently included in the IDL path cache, along with the number of .pro or .sav files found in those directories. See "New Path Caching" on page 17 for more details on path caching.

Enhancement to the MESSAGE Routine

The MESSAGE procedure now allows you to re-issue the most recent error, using the REISSUE_LAST keyword. By using this keyword in conjunction with the CATCH procedure, your code can catch an error caused by called code, perform recovery actions, and then reissue the error to your caller. See "MESSAGE" on page 109 for details.

Enhancement to the RESOLVE_ALL Routine

The RESOLVE_ALL procedure now allows you to specify a list of object class names via the new CLASS keyword. Class definition files for the specified classes and their superclasses are compiled, as are all methods of the specified classes and their superclasses. See "RESOLVE_ALL" on page 109 for details.

Enhancement to the SHMMAP Routine

The SHMMAP routine has been enhanced to allow creation of a private file mapping to a file for which the user has only read permission. See "SHMMAP" on page 110 for details.

Enhancement to the STRSPLIT Routine

The STRSPLIT function now allows you to obtain the number of matched substrings returned by STRSPLIT via the new COUNT keyword. See “STRSPLIT” on page 110 for details.

New ARRAY_INDICES Function

The new ARRAY_INDICES function converts one-dimensional subscripts of an array into corresponding multi-dimensional subscripts. See “ARRAY_INDICES” in the *IDL Reference Guide* manual for more details.

New IDL_VALIDNAME Function

The new IDL_VALIDNAME function determines whether a string may be used as a valid IDL identifier (e.g. variable name, structure tag name, etc.). See “IDL_VALIDNAME” in the *IDL Reference Guide* manual for more details.

File Access Enhancements

The following enhancements have been made in the area of File Access in the IDL 6.0 release:

- NetCDF Library Update
- Enhancement to the FILE_LINES Routine
- New FILE_BASENAME and FILE_DIRNAME Functions

NetCDF Library Update

IDL 6.0 has been upgraded to use NetCDF version 3.5. This library upgrade does not include new functionality added to the NetCDF code library since version 2.4. However, it does take advantage of performance improvements, bug fixes, and other such enhancements.

Enhancement to the FILE_LINES Routine

The FILE_LINES function now allows you to obtain the number of lines of text within a GZIP compressed file or files, using the COMPRESS keyword. If this keyword is set, FILE_LINES assumes the input files are compressed in the standard GZIP format, and decompresses the data to count the number of lines. See "FILE_LINES" on page 100 for details.

New FILE_BASENAME and FILE_DIRNAME Functions

Given a file path, the new FILE_BASENAME function returns the base file name it references, and the new FILE_DIRNAME function returns the directory part (i.e. all of the path except for the base file name). These functions are similar to, and based on, the standard Unix basename(1) and dirname(1) utilities.

See "FILE_BASENAME" and "FILE_DIRNAME" in the *IDL Reference Guide* manual for more details.

IDLDE Enhancements

The IDL Development Environment has been enhanced in the following ways for the 6.0 release:

- Path Cache Preference
- New Visualization Menu for iTools

Path Cache Preference

The Path tab of the Preferences dialog now allows you to enable or disable the IDL path cache mechanism.

New Visualization Menu for iTools

The new **Visualization** submenu to the **File** → **New** menu allows you to access the five new pre-built iTools for interactive plotting.

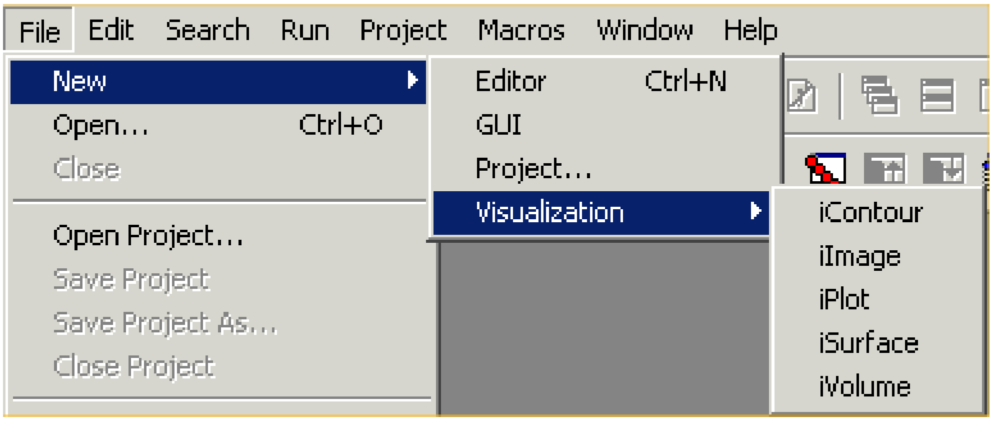


Figure 1-2: New Visualization Menu

See "New iTools for Interactive Analysis" on page 10 for more information on the pre-built iTools.

User Interface Toolkit Enhancements

The following enhancements have been made to IDL's UI toolkit for the 6.0 release to help you give your IDL applications more powerful and friendly user interfaces:

- Enhancements to the DIALOG_PICKFILE Routine
- Button Widget Enhancements
- New WIDGET_PROPERTYSHEET Function
- Enhancements to the WIDGET_CONTROL and WIDGET_INFO Routines

Enhancements to the DIALOG_PICKFILE Routine

The DIALOG_PICKFILE function now allows you to specify a default extension, with the DEFAULT_EXTENSION keyword. By setting this keyword to a scalar string representing the default extension, you can append the value returned by DIALOG_PICKFILE with this extension.

You can now also prompt users when using DIALOG_PICKFILE to attempt to overwrite files. By setting the OVERWRITE_PROMPT keyword along with the WRITE keyword, DIALOG_PICKFILE will automatically prompt the user with an overwrite dialog when a file that already exists is selected. See "DIALOG_PICKFILE" on page 99 for details.

Button Widget Enhancements

The PUSHBUTTON_EVENTS keyword has been added to WIDGET_BUTTON, allowing you to create buttons that generate separate widget events when the mouse button or space bar is pressed and released. See "WIDGET_BUTTON" on page 111 for details.

The PUSHBUTTON_EVENTS keyword has been added to WIDGET_CONTROL, allowing you to change the widget event generation properties of a button widget after creation. See "WIDGET_CONTROL" on page 112 for details.

The PUSHBUTTON_EVENTS keyword has been added to WIDGET_INFO, allowing you to query the pushbutton events setting of a specified button widget. See "WIDGET_INFO" on page 113 for details.

New WIDGET_PROPERTYSHEET Function

The new WIDGET_PROPERTYSHEET function creates a property sheet widget, which exposes the *properties* of an IDL object in a graphical interface.

For more details, see "WIDGET_PROPERTYSHEET" in the *IDL Reference Guide* manual.

Enhancements to the WIDGET_CONTROL and WIDGET_INFO Routines

The WIDGET_CONTROL procedure and the WIDGET_INFO function now allow access to the new property sheet widget.

By setting WIDGET_CONTROL's new REFRESH_PROPERTY keyword to a property identifier or array of identifiers, you can synchronize the identified properties with their values in a component. See "WIDGET_CONTROL" on page 112 for more details.

By setting WIDGET_INFO's new COMPONENT keyword to an object reference of a component, you can query specific components within a property sheet containing multiple components. By setting WIDGET_INFO's new PROPERTY_VALID keyword to a string, you can determine if that string is a valid identifier. If the identifier is valid, WIDGET_INFO's new PROPERTY_VALUE keyword can be set to this identifier to retrieve the value of the identified property within the property sheet. See "WIDGET_INFO" on page 113 for more details.

Enhancement to WIDGET_DROPLIST

The value of the list in a droplist widget can now be retrieved using the GET_VALUE keyword to WIDGET_CONTROL. The list values are returned as a scalar string or string array.

Documentation Enhancements

In addition to documentation for new and enhanced IDL features, the following enhancements to the IDL documentation set are included in the 6.0 release:

- New iTools User's Guide
- New iTools Developer's Guide

New iTools User's Guide

The new iTools are a set of interactive utilities that combine data analysis and visualization with the task of producing presentation quality graphics. Five iTools have been pre-built in IDL 6.0:

- iContour - for contour lines
- iImage - for image displays
- iPlot - for two and three dimensional plots
- iSurface - for surface representations
- iVolume - for volume visualizations

The *iTools User's Guide* walks you through calling these tools and using the iTool system interactively.

New iTools Developer's Guide

The new iTool system can be used to extend the pre-built tools with your own operations, manipulations, visualization types, and GUI controls. This system can also be used to create your own custom tools based on the iTools component framework. The *iTools Developer's Guide* instructs you on how to use the iTools component framework to develop your own iTools or build on existing ones.

New and Enhanced IDL Objects

This section describes the following:

- New IDL Object Classes
- New IDL Object Properties
- IDL Object Property Enhancements
- IDL Object Method Enhancements

Note

All the atomic graphical (IDLgr*) object classes are now subclasses of the new IDLitComponent object class.

New IDL Object Classes

The following table describes the new object classes in IDL 6.0 for Windows.

New Object Class	Description
IDLitCommand	The base functionality for the iTools command buffer system.
IDLitCommandSet	A container for IDLitCommand objects, which allows a group of commands to be managed as a single item.
IDLitComponent	A core or base component, from which all other components subclass. **Note -** This class is now a superclass of all the atomic graphical (IDLgr*) object classes.
IDLitContainer	A specialization of the IDL_Container class that manages a collection of IDLitComponents and provides methods for working with the Identifier system of the iTools framework.

New Object Class	Description
IDLitData	A generic data storage object that can hold any IDL data type available. It provides typing, metadata, and data change notification functionality. When coupled with IDLitDataContainer, it forms the element for the construction of composite data types.
IDLitDataContainer	A container for IDLitData and IDLitDataContainer objects. This container is used to form hierarchical data structures. Data and DataContainer objects can be added and removed to/from a DataContainer during program execution, allowing for dynamic creation of composite data types.
IDLitDataOperation	A subclass to IDLitOperation that automates data access and automatically records information for the undo-redo system.
IDLitIMessaging	An interface providing common methods to send or trigger messaging and error actions, which may occur during execution.
IDLitManipulator	The base functionality of the iTools manipulator system.
IDLitManipulatorContainer	A container for IDLitManipulator objects, which allows for the construction of manipulator hierarchies. This container implements the concept of a current manipulator for the items it contains.
IDLitManipulatorManager	A specialization of the manipulator container (IDLitManipulatorContainer), which acts as the root of the manipulator hierarchy.
IDLitManipulatorVisual	The basis of all visual elements associated with an interactive manipulator.

New Object Class	Description
IDLitOperation	The basis for all iTool operations. It defines how an operation is executed and how information about the operation is recorded for the command transaction (undo-redo) system.
IDLitParameter	An interface providing parameter management methods to associate parameter names with IDLitData objects.
IDLitParameterSet	A specialized subclass of the IDLitDataContainer class. This class provides the ability to associate names with contained IDLitData objects.
IDLitReader	The definition of the interface and the process used to construct file readers for the iTools framework. When a new file reader is constructed for the iTools system, a new class is subclassed from this IDLitReader class.
IDLitTool	All the functionality provided by a particular instance of an IDL Intelligent Tool (iTool). This object provides the management systems for the underlying tool functionality.
IDLitUI	A link between the underlying functionality of an iTool and the IDL widget interface.
IDLitVisualization	The basis for all iTool visualizations. All visualization components subclass from this class.
IDLitWindow	The basis for all iTool visualization windows. All iTool visualization windows subclass from this class.

New Object Class	Description
IDLitWriter	The definition of the interface and the process used to construct file writers for the iTools framework. When a new file writer is constructed for the iTools system, a new class is subclassed from this IDLitWriter class.
IDLjavaObject	An IDL object encapsulating a Java object. IDL provides data type and other translation services, allowing IDL programs to access the Java object's methods and properties using standard IDL syntax.

New IDL Object Properties

The following table describes new and updated properties to IDL objects.

IDLgrAxis

New Property	Description
DEPTH_TEST_DISABLE	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth testing. A model may also enable or disable depth testing. Set this property to 1 to explicitly disable depth buffer testing while drawing this object. Set this property to 2 to explicitly enable depth testing for this object. Disabling depth testing allows an object to draw itself on top of other objects already on the screen, even if the object is located behind them.

New Property	Description
DEPTH_TEST_FUNCTION	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always sets a depth test function of LESS. A model may also set a depth test function value. The graphics device compares the object's depth at a particular pixel location with the depth stored in the depth buffer at that same pixel location. If the comparison test passes, the object's pixel is drawn at that location on the screen and the depth buffer is updated (if depth writing is enabled). Set this property to any of the following values to use the desired function while rendering this object. • 0 = INHERIT - use the test function set for the parent model or view. • 1 = NEVER - never passes. • 2 = LESS - passes if the depth of the object's pixel is less than the depth buffer's value. • 3 = EQUAL - passes if the depth of the object's pixel is equal to the depth buffer's value. • 4 = LESS OR EQUAL - passes if the depth of the object's pixel is less than or equal to the depth buffer's value. • 5 = GREATER - passes if the depth of the object's pixel is greater than or equal to the depth buffer's value. • 6 = NOT EQUAL - passes if the depth of the object's pixel is not equal to the depth buffer's value. • 7 = GREATER OR EQUAL - passes if the depth of the object's pixel is greater than or equal to the depth buffer's value. • 8 = ALWAYS - always passes **Note -** Less means closer to the viewer.

New Property	Description
DEPTH_WRITE_DISABLE	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth writing. A model may also enable or disable depth writing. Set this property to 1 to explicitly disable depth buffer writing while rendering this object. Set this property to 2 to explicitly enable depth writing for this object. Disabling depth writing allows an object to be overdrawn by other objects, even if the object is located in front of them.

New Property	Description
REGISTER_PROPERTIES	Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered. For IDLgrAxis, the available properties and their iTool data types are: • COLOR (color) • DEPTH_TEST_DISABLE (enumerated list) • DEPTH_TEST_FUNCTION (enumerated list) • DEPTH_WRITE_DISABLE (enumerated list) • DIRECTION (enumerated list) • EXACT (Boolean) • EXTEND (Boolean) • GRIDSTYLE (linestyle) • HIDE (Boolean) • LOG (Boolean) • MAJOR (integer) • MINOR (integer) • NOTEXT (Boolean) • PALETTE (user-defined) • SUBTICKLEN (float) • TEXTPOS (user-defined) • THICK (thickness) • TICKDIR (enumerated list) • TICKINTERVAL (float) • TICKLAYOUT (enumerated list) • TICKLEN (float) • TICKUNITS (string)

IDLgrBuffer

New Property	Description
REGISTER_PROPERTIES	Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered. For IDLgrBuffer, the available properties and their iTool data types are: • COLOR_MODEL (enumerated list) • N_COLORS (integer) • PALETTE (user-defined) • QUALITY (enumerated list) • RESOLUTION (user-defined)

IDLgrClipboard

New Property	Description
REGISTER_PROPERTIES	Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered. For IDLgrClipboard, the available properties and their iTool data types are: • COLOR_MODEL (enumerated list) • N_COLORS (integer) • PALETTE (user-defined) • QUALITY (enumerated list) • RESOLUTION (user-defined)

IDLgrContour

New Property	Description
DEPTH_TEST_DISABLE	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth testing. A model may also enable or disable depth testing. Set this property to 1 to explicitly disable depth buffer testing while drawing this object. Set this property to 2 to explicitly enable depth testing for this object. Disabling depth testing allows an object to draw itself on top of other objects already on the screen, even if the object is located behind them.

New Property	Description
DEPTH_TEST_FUNCTION	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always sets a depth test function of LESS. A model may also set a depth test function value. The graphics device compares the object's depth at a particular pixel location with the depth stored in the depth buffer at that same pixel location. If the comparison test passes, the object's pixel is drawn at that location on the screen and the depth buffer is updated (if depth writing is enabled). Set this property to any of the following values to use the desired function while rendering this object. • 0 = INHERIT - use the test function set for the parent model or view. • 1 = NEVER - never passes. • 2 = LESS - passes if the depth of the object's pixel is less than the depth buffer's value. • 3 = EQUAL - passes if the depth of the object's pixel is equal to the depth buffer's value. • 4 = LESS OR EQUAL - passes if the depth of the object's pixel is less than or equal to the depth buffer's value. • 5 = GREATER - passes if the depth of the object's pixel is greater than or equal to the depth buffer's value. • 6 = NOT EQUAL - passes if the depth of the object's pixel is not equal to the depth buffer's value. • 7 = GREATER OR EQUAL - passes if the depth of the object's pixel is greater than or equal to the depth buffer's value. • 8 = ALWAYS - always passes **Note -** Less means closer to the viewer.

New Property	Description
DEPTH_WRITE_DISABLE	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth writing. A model may also enable or disable depth writing. Set this property to 1 to explicitly disable depth buffer writing while rendering this object. Set this property to 2 to explicitly enable depth writing for this object. Disabling depth writing allows an object to be overdrawn by other objects, even if the object is located in front of them.

New Property	Description
REGISTER_PROPERTIES	Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered. For IDLgrContour, the available properties and their iTool data types are: • ANISOTROPY (user-defined) • C_COLOR (user-defined) • C_FILL_PATTERN (user-defined) • C_LINESTYLE (user-defined) • C_THICK (user-defined) • C_VALUE (user-defined) • COLOR (color) • DEPTH_OFFSET (integer) • DEPTH_TEST_DISABLE (enumerated list) • DEPTH_TEST_FUNCTION (enumerated list) • DEPTH_WRITE_DISABLE (enumerated list) • DOWNHILL (enumerated list) • FILL (Boolean) • HIDE (Boolean) • MAX_VALUE (float) • MIN_VALUE (float) • N_LEVELS (integer) • PALETTE (user-defined) • PLANAR (enumerated list) • SHADING (enumerated list) • TICKINTERVAL (float) • TICKLEN (float)

IDLgrImage

New Property	Description
DEPTH_TEST_DISABLE	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth testing. A model may also enable or disable depth testing. Set this property to 1 to explicitly disable depth buffer testing while drawing this object. Set this property to 2 to explicitly enable depth testing for this object. Disabling depth testing allows an object to draw itself on top of other objects already on the screen, even if the object is located behind them.

New Property	Description
DEPTH_TEST_FUNCTION	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always sets a depth test function of LESS. A model may also set a depth test function value. The graphics device compares the object's depth at a particular pixel location with the depth stored in the depth buffer at that same pixel location. If the comparison test passes, the object's pixel is drawn at that location on the screen and the depth buffer is updated (if depth writing is enabled). Set this property to any of the following values to use the desired function while rendering this object. • 0 = INHERIT - use the test function set for the parent model or view. • 1 = NEVER - never passes. • 2 = LESS - passes if the depth of the object's pixel is less than the depth buffer's value. • 3 = EQUAL - passes if the depth of the object's pixel is equal to the depth buffer's value. • 4 = LESS OR EQUAL - passes if the depth of the object's pixel is less than or equal to the depth buffer's value. • 5 = GREATER - passes if the depth of the object's pixel is greater than or equal to the depth buffer's value. • 6 = NOT EQUAL - passes if the depth of the object's pixel is not equal to the depth buffer's value. • 7 = GREATER OR EQUAL - passes if the depth of the object's pixel is greater than or equal to the depth buffer's value. • 8 = ALWAYS - always passes **Note -** Less means closer to the viewer.

New Property	Description
DEPTH_WRITE_DISABLE	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth writing. A model may also enable or disable depth writing. Set this property to 1 to explicitly disable depth buffer writing while rendering this object. Set this property to 2 to explicitly enable depth writing for this object. Disabling depth writing allows an object to be overdrawn by other objects, even if the object is located in front of them.
REGISTER_PROPERTIES	Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered. For IDLgrImage, the available properties and their iTool data types are: • BLEND_FUNCTION (user-defined) • CHANNEL (integer) • COLOR (color) • DEPTH_TEST_DISABLE (enumerated list) • DEPTH_TEST_FUNCTION (enumerated list) • DEPTH_WRITE_DISABLE (enumerated list) • DIMENSIONS (user-defined) • GREYSCALE (Boolean) • HIDE (Boolean) • INTERLEAVE (enumerated list) • INTERPOLATE (enumerated list) • LOCATION (user-defined) • ORDER (enumerated list) • PALETTE (user-defined) • SUB_RECT (user-defined)

IDLgrLight

New Property	Description
REGISTER_PROPERTIES	Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered. For IDLgrLight, the available properties and their iTool data types are: • ATTENUATION (user-defined) • COLOR (color) • CONEANGLE (integer) • DIRECTION (user-defined) • FOCUS (float) • HIDE (Boolean) • INTENSITY (float) • LOCATION (user-defined) • PALETTE (user-defined) • TYPE (enumerated list)

IDLgrModel

New Property	Description
DEPTH_TEST_DISABLE	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth testing. Set this property to 1 to explicitly disable depth buffer testing while drawing all objects contained in this model. Set this property to 2 to explicitly enable depth testing for all objects contained in this model. Disabling depth testing allows an object to draw itself on top of other objects already on the screen, even if the object is located behind them. This value may be overridden by individual models or objects contained in this model.

New Property	Description
DEPTH_TEST_FUNCTION	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always sets a depth test function of LESS. The graphics device compares the object's depth at a particular pixel location with the depth stored in the depth buffer at that same pixel location. If the comparison test passes, the object's pixel is drawn at that location on the screen and the depth buffer is updated (if depth writing is enabled). This value may be overridden by individual models or objects contained in this model. Set this property to any of the following values to use the desired function while drawing all objects contained in this model. • 0 = INHERIT - use the test function set for the parent model or view. • 1 = NEVER - never passes. • 2 = LESS - passes if the depth of the object's pixel is less than the depth buffer's value. • 3 = EQUAL - passes if the depth of the object's pixel is equal to the depth buffer's value. • 4 = LESS OR EQUAL - passes if the depth of the object's pixel is less than or equal to the depth buffer's value. • 5 = GREATER - passes if the depth of the object's pixel is greater than or equal to the depth buffer's value. • 6 = NOT EQUAL - passes if the depth of the object's pixel is not equal to the depth buffer's value. • 7 = GREATER OR EQUAL - passes if the depth of the object's pixel is greater than or equal to the depth buffer's value. • 8 = ALWAYS - always passes **Note -** Less means closer to the viewer.

New Property	Description
DEPTH_WRITE_DISABLE	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth writing. Set this property to 1 to explicitly disable depth buffer writing while drawing the objects contained in this model. Set this property to 2 to explicitly enable depth writing for the objects contained in this model. Disabling depth writing allows an object to be overdrawn by other objects, even if the object is located in front of them.This value may be overridden by individual models or objects contained in this model.
REGISTER_PROPERTIES	Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered. For IDLgrModel, the available properties and their iTool data types are: • CLIP_PLANES (user-defined) • DEPTH_TEST_DISABLE (enumerated list) • DEPTH_TEST_FUNCTION (enumerated list) • DEPTH_WRITE_DISABLE (enumerated list) • HIDE (Boolean) • LIGHTING (enumerated list) • SELECT_TARGET (Boolean) • TRANSFORM (user-defined)

IDLgrPlot

New Property	Description
DEPTH_TEST_DISABLE	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth testing. A model may also enable or disable depth testing. Set this property to 1 to explicitly disable depth buffer testing while drawing this object. Set this property to 2 to explicitly enable depth testing for this object. Disabling depth testing allows an object to draw itself on top of other objects already on the screen, even if the object is located behind them.

New Property	Description
DEPTH_TEST_FUNCTION	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always sets a depth test function of LESS. A model may also set a depth test function value. The graphics device compares the object's depth at a particular pixel location with the depth stored in the depth buffer at that same pixel location. If the comparison test passes, the object's pixel is drawn at that location on the screen and the depth buffer is updated (if depth writing is enabled). Set this property to any of the following values to use the desired function while rendering this object. • 0 = INHERIT - use the test function set for the parent model or view. • 1 = NEVER - never passes. • 2 = LESS - passes if the depth of the object's pixel is less than the depth buffer's value. • 3 = EQUAL - passes if the depth of the object's pixel is equal to the depth buffer's value. • 4 = LESS OR EQUAL - passes if the depth of the object's pixel is less than or equal to the depth buffer's value. • 5 = GREATER - passes if the depth of the object's pixel is greater than or equal to the depth buffer's value. • 6 = NOT EQUAL - passes if the depth of the object's pixel is not equal to the depth buffer's value. • 7 = GREATER OR EQUAL - passes if the depth of the object's pixel is greater than or equal to the depth buffer's value. • 8 = ALWAYS - always passes **Note -** Less means closer to the viewer.

New Property	Description
DEPTH_WRITE_DISABLE	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth writing. A model may also enable or disable depth writing. Set this property to 1 to explicitly disable depth buffer writing while rendering this object. Set this property to 2 to explicitly enable depth writing for this object. Disabling depth writing allows an object to be overdrawn by other objects, even if the object is located in front of them.
REGISTER_PROPERTIES	Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered. For IDLgrPlot, the available properties and their iTool data types are: • COLOR (color) • DEPTH_TEST_DISABLE (enumerated list) • DEPTH_TEST_FUNCTION (enumerated list) • DEPTH_WRITE_DISABLE (enumerated list) • HIDE (Boolean) • HISTOGRAM (Boolean) • LINESTYLE (linestyle) • MAX_VALUE (float) • MIN_VALUE (float) • NSUM (integer) • PALETTE (user-defined) • POLAR (Boolean) • THICK (thickness) • VERT_COLORS (user-defined)

IDLgrPolygon

New Property	Description
DEPTH_TEST_DISABLE	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth testing. A model may also enable or disable depth testing. Set this property to 1 to explicitly disable depth buffer testing while drawing this object. Set this property to 2 to explicitly enable depth testing for this object. Disabling depth testing allows an object to draw itself on top of other objects already on the screen, even if the object is located behind them.

New Property	Description
DEPTH_TEST_FUNCTION	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always sets a depth test function of LESS. A model may also set a depth test function value. The graphics device compares the object's depth at a particular pixel location with the depth stored in the depth buffer at that same pixel location. If the comparison test passes, the object's pixel is drawn at that location on the screen and the depth buffer is updated (if depth writing is enabled). Set this property to any of the following values to use the desired function while rendering this object. • 0 = INHERIT **-** use the test function set for the parent model or view. • 1 = NEVER **-** never passes. • 2 = LESS **-** passes if the depth of the object's pixel is less than the depth buffer's value. • 3 = EQUAL **-** passes if the depth of the object's pixel is equal to the depth buffer's value. • 4 = LESS OR EQUAL **-** passes if the depth of the object's pixel is less than or equal to the depth buffer's value. • 5 = GREATER **-** passes if the depth of the object's pixel is greater than or equal to the depth buffer's value. • 6 = NOT EQUAL **-** passes if the depth of the object's pixel is not equal to the depth buffer's value. • 7 = GREATER OR EQUAL **-** passes if the depth of the object's pixel is greater than or equal to the depth buffer's value. • 8 = ALWAYS **-** always passes **Note -** Less means closer to the viewer.

New Property	Description
DEPTH_WRITE_DISABLE	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth writing. A model may also enable or disable depth writing. Set this property to 1 to explicitly disable depth buffer writing while rendering this object. Set this property to 2 to explicitly enable depth writing for this object. Disabling depth writing allows an object to be overdrawn by other objects, even if the object is located in front of them.

New Property	Description
REGISTER_PROPERTIES	Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered. For IDLgrPolygon, the available properties and their iTool data types are: • BOTTOM (color) • COLOR (color) • DEPTH_OFFSET (integer) • DEPTH_TEST_DISABLE (enumerated list) • DEPTH_TEST_FUNCTION (enumerated list) • DEPTH_WRITE_DISABLE (enumerated list) • HIDDEN_LINE (Boolean) • HIDE (Boolean) • LINESTYLE (linestyle) • PALETTE (user-defined) • REJECT (enumerated list) • SHADING (enumerated list) • STYLE (enumerated list) • TEXTURE_INTERP (enumerated list) • TEXTURE_MAP (user-defined) • THICK (thickness) • VERT_COLORS (user-defined) • ZERO_OPACITY_SKIP (Boolean)

IDLgrPolyline

New Property	Description
DEPTH_TEST_DISABLE	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth testing. A model may also enable or disable depth testing. Set this property to 1 to explicitly disable depth buffer testing while drawing this object. Set this property to 2 to explicitly enable depth testing for this object. Disabling depth testing allows an object to draw itself on top of other objects already on the screen, even if the object is located behind them.

New Property	Description
DEPTH_TEST_FUNCTION	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always sets a depth test function of LESS. A model may also set a depth test function value. The graphics device compares the object's depth at a particular pixel location with the depth stored in the depth buffer at that same pixel location. If the comparison test passes, the object's pixel is drawn at that location on the screen and the depth buffer is updated (if depth writing is enabled). Set this property to any of the following values to use the desired function while rendering this object. • 0 = INHERIT **-** use the test function set for the parent model or view. • 1 = NEVER **-** never passes. • 2 = LESS **-** passes if the depth of the object's pixel is less than the depth buffer's value. • 3 = EQUAL **-** passes if the depth of the object's pixel is equal to the depth buffer's value. • 4 = LESS OR EQUAL **-** passes if the depth of the object's pixel is less than or equal to the depth buffer's value. • 5 = GREATER **-** passes if the depth of the object's pixel is greater than or equal to the depth buffer's value. • 6 = NOT EQUAL **-** passes if the depth of the object's pixel is not equal to the depth buffer's value. • 7 = GREATER OR EQUAL **-** passes if the depth of the object's pixel is greater than or equal to the depth buffer's value. • 8 = ALWAYS **-** always passes **Note -** Less means closer to the viewer.

New Property	Description
DEPTH_WRITE_DISABLE	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth writing. A model may also enable or disable depth writing. Set this property to 1 to explicitly disable depth buffer writing while rendering this object. Set this property to 2 to explicitly enable depth writing for this object. Disabling depth writing allows an object to be overdrawn by other objects, even if the object is located in front of them.
REGISTER_PROPERTIES	Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered. For IDLgrPolyline, the available properties and their iTool data types are: • COLOR (color) • DEPTH_TEST_DISABLE (enumerated list) • DEPTH_TEST_FUNCTION (enumerated list) • DEPTH_WRITE_DISABLE (enumerated list) • HIDE (Boolean) • LINESTYLE (linestyle) • PALETTE (user-defined) • SHADING (enumerated list) • THICK (thickness) • VERT_COLORS (user-defined)

IDLgrPrinter

New Property	Description
REGISTER_PROPERTIES	Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered. For IDLgrPrinter, the available properties and their iTool data types are: • COLOR_MODEL (enumerated list) • N_COLORS (integer) • PALETTE (user-defined) • QUALITY (enumerated list) • RESOLUTION (float)

IDLgrROI

New Property	Description
DEPTH_TEST_DISABLE	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth testing. A model may also enable or disable depth testing. Set this property to 1 to explicitly disable depth buffer testing while drawing this object. Set this property to 2 to explicitly enable depth testing for this object. Disabling depth testing allows an object to draw itself on top of other objects already on the screen, even if the object is located behind them.

New Property	Description
DEPTH_TEST_FUNCTION	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always sets a depth test function of LESS. A model may also set a depth test function value. The graphics device compares the object's depth at a particular pixel location with the depth stored in the depth buffer at that same pixel location. If the comparison test passes, the object's pixel is drawn at that location on the screen and the depth buffer is updated (if depth writing is enabled). Set this property to any of the following values to use the desired function while rendering this object. • 0 = INHERIT - use the test function set for the parent model or view. • 1 = NEVER - never passes. • 2 = LESS - passes if the depth of the object's pixel is less than the depth buffer's value. • 3 = EQUAL - passes if the depth of the object's pixel is equal to the depth buffer's value. • 4 = LESS OR EQUAL - passes if the depth of the object's pixel is less than or equal to the depth buffer's value. • 5 = GREATER - passes if the depth of the object's pixel is greater than or equal to the depth buffer's value. • 6 = NOT EQUAL - passes if the depth of the object's pixel is not equal to the depth buffer's value. • 7 = GREATER OR EQUAL - passes if the depth of the object's pixel is greater than or equal to the depth buffer's value. • 8 = ALWAYS - always passes **Note -** Less means closer to the viewer.

New Property	Description
DEPTH_WRITE_DISABLE	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth writing. A model may also enable or disable depth writing. Set this property to 1 to explicitly disable depth buffer writing while rendering this object. Set this property to 2 to explicitly enable depth writing for this object. Disabling depth writing allows an object to be overdrawn by other objects, even if the object is located in front of them.
REGISTER_PROPERTIES	Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered. For IDLgrROI, the available properties and their iTool data types are: • COLOR (color) • DEPTH_TEST_DISABLE (enumerated list) • DEPTH_TEST_FUNCTION (enumerated list) • DEPTH_WRITE_DISABLE (enumerated list) • HIDE (Boolean) • LINESTYLE (linestyle) • PALETTE (user-defined) • THICK (thickness)

IDLgrROIGroup

New Property	Description
DEPTH_TEST_DISABLE	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth testing. A model may also enable or disable depth testing. Set this property to 1 to explicitly disable depth buffer testing while drawing this object. Set this property to 2 to explicitly enable depth testing for this object. Disabling depth testing allows an object to draw itself on top of other objects already on the screen, even if the object is located behind them.

New Property	Description
DEPTH_TEST_FUNCTION	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always sets a depth test function of LESS. A model may also set a depth test function value. The graphics device compares the object's depth at a particular pixel location with the depth stored in the depth buffer at that same pixel location. If the comparison test passes, the object's pixel is drawn at that location on the screen and the depth buffer is updated (if depth writing is enabled). Set this property to any of the following values to use the desired function while rendering this object. • 0 = INHERIT - use the test function set for the parent model or view. • 1 = NEVER - never passes. • 2 = LESS - passes if the depth of the object's pixel is less than the depth buffer's value. • 3 = EQUAL - passes if the depth of the object's pixel is equal to the depth buffer's value. • 4 = LESS OR EQUAL - passes if the depth of the object's pixel is less than or equal to the depth buffer's value. • 5 = GREATER - passes if the depth of the object's pixel is greater than or equal to the depth buffer's value. • 6 = NOT EQUAL - passes if the depth of the object's pixel is not equal to the depth buffer's value. • 7 = GREATER OR EQUAL - passes if the depth of the object's pixel is greater than or equal to the depth buffer's value. • 8 = ALWAYS - always passes **Note -** Less means closer to the viewer.

New Property	Description
DEPTH_WRITE_DISABLE	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth writing. A model may also enable or disable depth writing. Set this property to 1 to explicitly disable depth buffer writing while rendering this object. Set this property to 2 to explicitly enable depth writing for this object. Disabling depth writing allows an object to be overdrawn by other objects, even if the object is located in front of them.
REGISTER_PROPERTIES	Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered. For IDLgrROIGroup, the available properties and their iTool data types are: • COLOR (color) • DEPTH_TEST_DISABLE (enumerated list) • DEPTH_TEST_FUNCTION (enumerated list) • DEPTH_WRITE_DISABLE (enumerated list) • HIDE (Boolean)

IDLgrScene

New Property	Description
REGISTER_PROPERTIES	Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered. For IDLgrScene, the available properties and their iTool data types are: • COLOR (color) • HIDE (Boolean) • TRANSPARENT (Boolean)

IDLgrSurface

New Property	Description
DEPTH_TEST_DISABLE	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth testing. A model may also enable or disable depth testing. Set this property to 1 to explicitly disable depth buffer testing while drawing this object. Set this property to 2 to explicitly enable depth testing for this object. Disabling depth testing allows an object to draw itself on top of other objects already on the screen, even if the object is located behind them.

New Property	Description
DEPTH_TEST_FUNCTION	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always sets a depth test function of LESS. A model may also set a depth test function value. The graphics device compares the object's depth at a particular pixel location with the depth stored in the depth buffer at that same pixel location. If the comparison test passes, the object's pixel is drawn at that location on the screen and the depth buffer is updated (if depth writing is enabled). Set this property to any of the following values to use the desired function while rendering this object. • 0 = INHERIT - use the test function set for the parent model or view. • 1 = NEVER - never passes. • 2 = LESS - passes if the depth of the object's pixel is less than the depth buffer's value. • 3 = EQUAL - passes if the depth of the object's pixel is equal to the depth buffer's value. • 4 = LESS OR EQUAL - passes if the depth of the object's pixel is less than or equal to the depth buffer's value. • 5 = GREATER - passes if the depth of the object's pixel is greater than or equal to the depth buffer's value. • 6 = NOT EQUAL - passes if the depth of the object's pixel is not equal to the depth buffer's value. • 7 = GREATER OR EQUAL - passes if the depth of the object's pixel is greater than or equal to the depth buffer's value. • 8 = ALWAYS - always passes **Note -** Less means closer to the viewer.

New Property	Description
DEPTH_WRITE_DISABLE	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth writing. A model may also enable or disable depth writing. Set this property to 1 to explicitly disable depth buffer writing while rendering this object. Set this property to 2 to explicitly enable depth writing for this object. Disabling depth writing allows an object to be overdrawn by other objects, even if the object is located in front of them.

New Property	Description
REGISTER_PROPERTIES	Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered. For IDLgrSurface, the available properties and their iTool data types are: • BOTTOM (color) • COLOR (color) • DEPTH_OFFSET (integer) • DEPTH_TEST_DISABLE (enumerated list) • DEPTH_TEST_FUNCTION (enumerated list) • DEPTH_WRITE_DISABLE (enumerated list) • EXTENDED_LEGO (Boolean) • HIDDEN_LINES (Boolean) • HIDE (Boolean) • LINESTYLE (linestyle) • MAX_VALUE (float) • MIN_VALUE (float) • PALETTE (user-defined) • SHADING (enumerated list) • SHOW_SKIRT (Boolean) • SKIRT (float) • STYLE (enumerated list) • TEXTURE_HIRES (Boolean) • TEXTURE_INTERP (enumerated list) • TEXTURE_MAP (user-defined) • THICK (thickness) • USE_TRIANGLES (enumerated list) • VERT_COLORS (user-defined) • ZERO_OPACITY_SKIP (Boolean)

IDLgrText

New Property	Description
ALPHA_CHANNEL	Set this property to a value in the range [0.0, 1.0] (1.0 is the default) to draw the text foreground and background with the specified blending factor. A value of 1.0 draws the text opaquely without blending the text with objects already drawn on the destination. Edges of the glyphs are always blended. A value of 0.0 draws no text at all. A value in the middle of the range draws the text semi-transparently, which provides a way of creating labels that are visible while allowing features blocked by the labels to still be seen. This property is used only when the RENDER_METHOD in effect is 0 (Texture).
DEPTH_TEST_DISABLE	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth testing. A model may also enable or disable depth testing. Set this property to 1 to explicitly disable depth buffer testing while drawing this object. Set this property to 2 to explicitly enable depth testing for this object. Disabling depth testing allows an object to draw itself on top of other objects already on the screen, even if the object is located behind them.

New Property	Description
DEPTH_TEST_FUNCTION	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always sets a depth test function of LESS. A model may also set a depth test function value. The graphics device compares the object's depth at a particular pixel location with the depth stored in the depth buffer at that same pixel location. If the comparison test passes, the object's pixel is drawn at that location on the screen and the depth buffer is updated (if depth writing is enabled). Set this property to any of the following values to use the desired function while rendering this object. • 0 = INHERIT - use the test function set for the parent model or view. • 1 = NEVER - never passes. • 2 = LESS - passes if the depth of the object's pixel is less than the depth buffer's value. • 3 = EQUAL - passes if the depth of the object's pixel is equal to the depth buffer's value. • 4 = LESS OR EQUAL - passes if the depth of the object's pixel is less than or equal to the depth buffer's value. • 5 = GREATER - passes if the depth of the object's pixel is greater than or equal to the depth buffer's value. • 6 = NOT EQUAL - passes if the depth of the object's pixel is not equal to the depth buffer's value. • 7 = GREATER OR EQUAL - passes if the depth of the object's pixel is greater than or equal to the depth buffer's value. • 8 = ALWAYS - always passes **Note -** Less means closer to the viewer.

New Property	Description
DEPTH_WRITE_DISABLE	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth writing. A model may also enable or disable depth writing. Set this property to 1 to explicitly disable depth buffer writing while rendering this object. Set this property to 2 to explicitly enable depth writing for this object. Disabling depth writing allows an object to be overdrawn by other objects, even if the object is located in front of them.
FILL_BACKGROUND	Set this property to zero (the default) to render the text with a transparent bitmap background, allowing graphics behind the text to show through between the glyphs. Set this property to non-zero to draw the text bitmap background with the color specified by the FILL_COLOR property. This property can only be used when RENDER_METHOD is set to 0 (Texture).
FILL_COLOR	Set this property to an RGB color vector or color index value to specify that the text bitmap background should be drawn using the specified color. This property is used only when the FILL_BACKGROUND property has a non-zero value and the RENDER_METHOD in effect is 0 (Texture). Set this property to -1 (the default) to specify that the text background should be drawn using the current view background color.

New Property	Description
KERNING	Set this property to a non-zero value (the default is zero) to enable kerning while rendering characters. Kerning reduces the amount of space between glyphs if the shape of each glyph allows it, according to the font information stored in the font’s file. For example, the letters "A" and "V" placed together, "AV", contains space that can be reduced by kerning. Enabling kerning may not necessarily result in rendering glyphs more closely together because some fonts do not contain the required kerning information. This property is used only when RENDER_METHOD is 0 (Texture).

New Property	Description
REGISTER_PROPERTIES	Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered. For IDLgrText, the available properties and their iTool data types are: • ALIGNMENT (float) • ALPHA_CHANNEL (float) • BASELINE (user-defined) • COLOR (color) • DEPTH_TEST_DISABLE (enumerated list) • DEPTH_TEST_FUNCTION (enumerated list) • DEPTH_WRITE_DISABLE (enumerated list) • FILL_BACKGROUND (Boolean) • FILL_COLOR (color) • HIDE (Boolean) • KERNING (Boolean) • LOCATIONS (user-defined) • ONGLASS (Boolean) • PALETTE (user-defined) • RECOMPUTE_DIMENSIONS (enumerated list) • RENDER_METHOD (enumerated list) • STRINGS (user-defined) • UPDIR (user-defined) • VERTICAL_ALIGNMENT (float)

New Property	Description
RENDER_METHOD	Set this property to one of the following values: • 0 (zero) TEXTURE - IDL renders the text by placing a bitmap representation of a glyph into a texture map and then rendering a polygon with the texture map. How the background portions of the texture map are drawn and how the entire texture map is blended into the scene are controlled by the ALPHA_CHANNEL, FILL_COLOR, and FILL_BACKGROUND properties. Leaving these three properties set to their default values produces a result that closely approximates the TRIANGLES rendering method. One important difference is that the glyph bitmaps are generated by the FreeType font rendering library, producing glyphs that are more accurately rendered and anti-aliased than those drawn with the TRIANGLES method. The TEXTURE method cannot be used on indexed color destinations. The text is rendered with the TRIANGLES method if the destination uses indexed color. • 1 (one) TRIANGLES - IDL renders the text by tessellating the glyph outline into a set of small triangles that are then drawn to produce the solid glyph. IDL also draws a blended line around the edge of the glyph to approximate anti-aliasing. This setting used to be the default behavior for IDL versions prior to IDL 6.0. **Note -** If IDLgrClipboard or IDLgrPrinter is drawn in vector mode (VECTOR = 1), any IDLgrText objects in the display are drawn as if the RENDER_METHOD property was set to 1 (TRIANGLES).

IDLgrView

New Property	Description
REGISTER_PROPERTIES	Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered. For IDLgrView, the available properties and their iTool data types are: • COLOR (color) • DEPTH_CUE (user-defined) • DOUBLE (Boolean) • EYE (float) • HIDE (enumerated list) • LOCATION (user-defined) • PROJECTION (enumerated list) • TRANSPARENT (Boolean) • UNITS (enumerated list) • VIEWPLANE_RECT (user-defined) • ZCLIP (user-defined)

IDLgrViewgroup

New Property	Description
REGISTER_PROPERTIES	Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered. For IDLgrViewgroup, the available properties and their iTool data types are: • HIDE (Boolean)

IDLgrVolume

New Property	Description
DEPTH_TEST_DISABLE	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth testing. A model may also enable or disable depth testing. Set this property to 1 to explicitly disable depth buffer testing while drawing this object. Set this property to 2 to explicitly enable depth testing for this object. Disabling depth testing allows an object to draw itself on top of other objects already on the screen, even if the object is located behind them.

New Property	Description
DEPTH_TEST_FUNCTION	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always sets a depth test function of LESS. A model may also set a depth test function value. The graphics device compares the object's depth at a particular pixel location with the depth stored in the depth buffer at that same pixel location. If the comparison test passes, the object's pixel is drawn at that location on the screen and the depth buffer is updated (if depth writing is enabled). Set this property to any of the following values to use the desired function while rendering this object. • 0 = INHERIT - use the test function set for the parent model or view. • 1 = NEVER - never passes. • 2 = LESS - passes if the depth of the object's pixel is less than the depth buffer's value. • 3 = EQUAL - passes if the depth of the object's pixel is equal to the depth buffer's value. • 4 = LESS OR EQUAL - passes if the depth of the object's pixel is less than or equal to the depth buffer's value. • 5 = GREATER - passes if the depth of the object's pixel is greater than or equal to the depth buffer's value. • 6 = NOT EQUAL - passes if the depth of the object's pixel is not equal to the depth buffer's value. • 7 = GREATER OR EQUAL - passes if the depth of the object's pixel is greater than or equal to the depth buffer's value. • 8 = ALWAYS - always passes **Note -** Less means closer to the viewer.

New Property	Description
DEPTH_WRITE_DISABLE	Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth writing. A model may also enable or disable depth writing. Set this property to 1 to explicitly disable depth buffer writing while rendering this object. Set this property to 2 to explicitly enable depth writing for this object. Disabling depth writing allows an object to be overdrawn by other objects, even if the object is located in front of them.

New Property	Description
REGISTER_PROPERTIES	Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered. For IDLgrVolume, the available properties (and their iTool data types) are: • AMBIENT (color) • BOUNDS (user-defined) • COMPOSITE_FUNCTION (enumerated list) • CUTTING_PLANES (user-defined) • DEPTH_CUE (user-defined) • DEPTH_TEST_DISABLE (enumerated list) • DEPTH_TEST_FUNCTION (enumerated list) • DEPTH_WRITE_DISABLE (enumerated list) • HIDE (enumerated list) • HINTS (enumerated list) • INTERPOLATE (enumerated list) • LIGHTING_MODEL (Boolean) • PALETTE (user-defined) • RENDER_STEP (user-defined) • TWO_SIDED (enumerated list) • ZBUFFER (Boolean) • ZERO_OPACITY_SKIP (Boolean)

IDLgrVRML

New Property	Description
REGISTER_PROPERTIES	Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered. For IDLgrVRML, the available properties and their iTool data types are: • COLOR_MODEL (enumerated list) • N_COLORS (integer) • PALETTE (user-defined) • QUALITY (enumerated list) • RESOLUTION (user-defined)

IDLgrWindow

New Property	Description
REGISTER_PROPERTIES	Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered. For IDLgrWindow, the available properties and their iTool data types are: • COLOR_MODEL (enumerated list) • CURRENT_ZOOM (float) • DIMENSIONS (user-defined) • DISPLAY_NAME (string) • LOCATION (user-defined) • N_COLORS (integer) • PALETTE (user-defined) • QUALITY (enumerated list) • RENDERER (enumerated list) • RESOLUTION (float) • RETAIN (enumerated list) • TITLE (string) • UNITS (enumerated list) • VIRTUAL_DIMENSIONS (user-defined) • VISIBLE_LOCATION (user-defined)
VIRTUAL_DIMENSIONS	Set this property to a two-element vector, [*width*, *height*], specifying the dimensions of the virtual canvas for this window. The default is [0, 0], indicating that the virtual canvas dimensions should match the visible dimensions (as specified via the DIMENSIONS keyword).

New Property	Description
VISIBLE_LOCATION	Set this property to a two-element vector, [*x*, *y*], specifying the lower left location of the visible portion of the canvas (relative to the virtual canvas).

IDL Object Property Enhancements

The following table describes updated properties to IDL object classes.

Note

The following table contains an update to the documentation for the RECOMPUTE_DIMENSIONS property of the IDLgrText object class. This property in IDL 6.0 has not functionally changed from previous versions of IDL.

IDLgrText

Item	Description
RECOMPUTE_DIMENSIONS	Set this property to one of the following values: • 0 = The physical size of the text is affected by model and view transforms. The size of the text in terms of data units is obtained from CHAR_DIMENSIONS. Since the character dimensions are specified in data units, the text will maintain the data space size specified by CHAR_DIMENSIONS as the transforms change. In other words, the physical text size changes along with other primitives. If the value of this property is [0, 0], the text font's point size is used to compute the physical size of the text in terms of data units using the transforms in effect for the first draw. This setting is the default value for this property. • 1 = The physical size of the text is only affected by model transforms. The CHAR_DIMENSIONS property is ignored. The size of the text is computed from the font's point size the first time it is drawn, and IDL does not try to keep the size of the text constant with respect to changes in the model transforms. • 2 = The physical size of the text is held constant, even as the model and view change. The CHAR_DIMENSIONS property is ignored and the text is always drawn with a physical size equal to the text font's point size. IDL adjusts its internal text transforms to maintain the physical size of the text.

IDL Object Method Enhancements

The following table describes new and updated keywords and arguments to IDL object methods.

IDLanROI::ComputeMask

Item	Description
PIXEL_CENTER	Set this keyword to a 2-element vector, [*x*, *y*], to indicate where the lower-left mask pixel is centered relative to a Cartesian grid. The default value is [0.0, 0.0], indicating that the lower-left pixel is centered at [0.0, 0.0].

New and Enhanced IDL Routines

This section describes the following:

- New IDL Routines
- IDL Routine Enhancements

New IDL Routines

The following is a list of new functions and procedures added to IDL in this release.

New Routine	Description
ARRAY_INDICES	Converts one-dimensional subscripts of an array into corresponding multi-dimensional subscripts.
FILE_BASENAME	Returns the *basename* of a file path. The basename is the final rightmost segment of the file path.
FILE_DIRNAME	Returns the *dirname* of a file path. The dirname is all of the file path except for the final rightmost segment of the file path.
ICONTOUR	Creates an iTool and associated user interface (UI) configured to display and manipulate contour data.
IDL_VALIDNAME	Determines whether a string may be used as a valid IDL variable name or structure tag name.
IDLITSYS_CREATETOOL	Creates an instance of the specified tool registered within the iTools system.
IIMAGE	Creates an iTool and associated user interface (UI) configured to display and manipulate image data.
IPLOT	Creates an iTool and associated user interface (UI) configured to display and manipulate plot data.

New Routine	Description
ISURFACE	Creates an iTool and associated user interface (UI) configured to display and manipulate surface data.
ITCURRENT	Set the current tool in the iTools system.
ITDELETE	Deletes a tool in the iTools system.
ITGETCURRENT	Gets the identifier of the current tool in the iTools system.
ITREGISTER	Registers tool object classes with the iTools system.
ITRESET	Resets the iTools session.
IVOLUME	Creates an iTool and associated user interface (UI) configured to display and manipulate volume data.
LOGICAL_AND	Performs a logical AND operation on its arguments, which can be scalar or array.
LOGICAL_OR	Performs a logical OR operation on its arguments, which can be scalar or array.
LOGICAL_TRUE	Determines whether its argument, which can be scalar or array, is non-zero (or non-NULL).
PATH_CACHE	Controls IDL's path caching mechanism.
WIDGET_PROPERTYSHEET	Creates a property sheet widget, which exposes the properties of an IDL object in a graphical interface.

IDL Routine Enhancements

The following is a list of new and updated keywords, arguments, and/or return values to existing IDL routines.

CURVEFIT

Keyword or item	Description
FITA	Set this keyword to a vector, with as many elements as *A*, which contains a zero for each fixed parameter, and a non-zero value for elements of *A* to fit. If not supplied, all parameters are taken to be non-fixed.
STATUS	Possible return values for STATUS are: • 0 = The computation was successful. • 1 = The computation failed. Chi-square was increasing without bounds. • 2 = The computation failed to converge in ITMAX iterations.

DIALOG_PICKFILE

Keyword or item	Description
DEFAULT_EXTENSION	Set this keyword to a scalar string representing the default extension to be appended onto the returned file name or names. If the returned file name already has an extension, then the value set for this keyword is not appended. The value for this keyword should not include the period (.). **Note -** This keyword only applies to file names typed into the dialog. This keyword does not apply to files selected within the dialog.

Keyword or item	Description
OVERWRITE_PROMPT	If this keyword is set along with the WRITE keyword and the user selects a file that already exists, then a dialog will be displayed asking if the user wants to replace the existing file or not. For multiple selections, the user is prompted separately for each file. If the user selects **No** then the user is returned to the file selection dialog; if the user selects **Yes** then the selection is allowed. This keyword has no effect unless the WRITE keyword is also set.

FILE_LINES

Keyword or item	Description
COMPRESS	If this keyword is set, FILE_LINES assumes that the files specified in *Path* contain data compressed in the standard GZIP format, and decompresses the data in order to count the number of lines. See the description of the COMPRESS keyword to the OPEN procedure for additional information.

GAUSSFIT

Keyword or item	Description
MEASURE_ERRORS	Set this keyword to a vector containing standard measurement errors for each point in the *Y* input argument. This vector must be the same length as the input arguments, *X* and *Y*. **Note -** For Gaussian errors (for example, instrumental uncertainties), MEASURE_ERRORS should be set to the standard deviations of each point in *Y*. For Poisson or statistical weighting, MEASURE_ERRORS should be set to SQRT(*Y*).

HELP

Keyword or item	Description
PATH_CACHE	Set this keyword to display a list of directories currently included in the IDL path cache, along with the number of .pro or .sav files found in those directories. See “PATH_CACHE” in the *IDL Reference Guide* manual for details.

INTERVAL_VOLUME

Keyword or item	Description

PROGRESS_CALLBACK	Set this keyword to a scalar string containing the name of the IDL function that the INTERVAL_VOLUME procedure calls at PROGRESS_PERCENT intervals as it generates the interval volume.
	
	The PROGRESS_CALLBACK function returns a zero to signal INTERVAL_VOLUME to stop generating the interval volume. This causes INTERVAL_VOLUME to return a single vertex and a connectivity array of [-1], which specifies an empty mesh. If the callback function returns any non-zero value, INTERVAL_VOLUME continues to generate the interval volume.
	
	The PROGRESS_CALLBACK function must specify a single argument, *Percent*, which INTERVAL_VOLUME sets to an integer between 0 and 100, inclusive.
	
	The PROGRESS_CALLBACK function may specify an optional USERDATA keyword parameter, which INTERVAL_VOLUME sets to the variable provided in the PROGRESS_USERDATA keyword.
	
	The following code shows an example of a progress callback function:
	
	`FUNCTION myProgressCallback, $`
	`   percent, USERDATA = myStruct`
	
	`oProgressBar = myStruct.oProgressBar`
	
	`; This method updates the progress bar`
	`; graphic and returns TRUE if the user`
	`; has NOT pressed the cancel button.`
	`keepGoing = oProgressBar -> $`
	`   UpdateProgressValue(percent)`
	
	`RETURN, keepGoing`
	
	`END`

Keyword or item	Description

PROGRESS_METHOD	Set this keyword to a scalar string containing the name of a function method that the INTERVAL_VOLUME procedure calls at PROGRESS_PERCENT intervals as it generates the interval volume. If this keyword is set, then the PROGRESS_OBJECT keyword must be set to an object reference that is an instance of a class that defines the specified method.
	
	The PROGRESS_METHOD function method callback has the same specification as the callback described in the PROGRESS_CALLBACK keyword, except that it is defined as an object class method:
	
	`FUNCTION myClass::myProgressCallback, $`
	`   percent, USERDATA = myStruct`

PROGRESS_OBJECT	Set this keyword to an object reference that is an instance of a class that defines the method specified with the PROGRESS_METHOD keyword. If this keyword is set, then the PROGRESS_METHOD keyword must also be set.
PROGRESS_PERCENT	Set this keyword to a scalar in the range [1, 100] to specify the interval between invocations of the callback function. The default value is 5 and IDL silently clamps other values to the range [1, 100]. For example, a value of 5 (the default) specifies INTERVAL_VOLUME will call the callback function when the interval volume process is 0% complete, 5% complete, 10% complete, ..., 95% complete, and 100% complete.
PROGRESS_USERDATA	Set this property to any IDL variable that INTERVAL_VOLUME passes to the callback function in the callback function's USERDATA keyword parameter. If this keyword is specified, then the callback function must be able to accept keyword parameters.

ISOSURFACE

Keyword or item	Description

PROGRESS_CALLBACK	Set this keyword to a scalar string containing the name of the IDL function that ISOSURFACE calls at PROGRESS_PERCENT intervals as it generates the isosurface.
	
	The PROGRESS_CALLBACK function returns a zero to signal ISOSURFACE to stop generating the isosurface. This causes ISOSURFACE to return a single vertex and a connectivity array of [-1], which specifies an empty polygon. If the callback function returns any non-zero value, ISOSURFACE continues to generate the isosurface.
	
	The PROGRESS_CALLBACK function must specify a single argument, *Percent*, which ISOSURFACE sets to an integer between 0 and 100, inclusive.
	
	The PROGRESS_CALLBACK function may specify an optional USERDATA keyword parameter, which ISOSURFACE sets to the variable provided in the PROGRESS_USERDATA keyword.
	
	The following code shows an example of a progress callback function:
	
	`FUNCTION myProgressCallback, percent,$`
	`   USERDATA = myStruct`
	
	`oProgressBar = myStruct.oProgressBar`
	
	`; This method updates the progress bar`
	`; graphic and returns TRUE if the user has`
	`; NOT pressed the cancel button.`
	`keepGoing = oProgressBar -> $`
	`   UpdateProgressValue(percent)`
	
	`RETURN, keepGoing`
	
	`END`

Keyword or item	Description

PROGRESS_METHOD	Set this keyword to a scalar string containing the name of a function method that ISOSURFACE calls at PROGRESS_PERCENT intervals as it generates the isosurface. If this keyword is set, then the PROGRESS_OBJECT keyword must be set to an object reference that is an instance of a class that defines the specified method.
	
	The PROGRESS_METHOD function method callback has the same specification as the callback described in the PROGRESS_CALLBACK keyword, except that it is defined as an object class method:
	
	`FUNCTION myClass::myProgressCallback, $`
	`   percent, USERDATA = myStruct`

PROGRESS_OBJECT	Set this keyword to an object reference that is an instance of a class that defines the method specified with the PROGRESS_METHOD keyword. If this keyword is set, then the PROGRESS_METHOD keyword must also be set.
PROGRESS_PERCENT	Set this keyword to a scalar in the range [1, 100] to specify the interval between invocations of the callback function. The default value is 5 and IDL silently clamps other values to the range [1, 100]. For example, a value of 5 (the default) specifies ISOSURFACE will call the callback function when the isosurface process is 0% complete, 5% complete, 10% complete, ..., 95% complete, and 100% complete.

Keyword or item	Description
PROGRESS_USERDATA	Set this property to any IDL variable that ISOSURFACE passes to the callback function in the callback function's USERDATA keyword parameter. If this keyword is specified, then the callback function must be able to accept keyword parameters.

LMGR

Keyword or item	Description
VM	Set this keyword to test whether the current IDL session is running in Virtual Machine mode. Virtual Machine applications do not provide access to the IDL Command Line.

MESH_DECIMATE

Keyword or item	Description

PROGRESS_CALLBACK	Set this keyword to a scalar string containing the name of the IDL function that MESH_DECIMATE calls at PROGRESS_PERCENT intervals as it decimates the polygonal mesh.
	
	The PROGRESS_CALLBACK function returns a zero to signal MESH_DECIMATE to stop decimating, which causes MESH_DECIMATE to return the partially decimated mesh. If the callback function returns a non-zero, MESH_DECIMATE continues to decimate the mesh.
	
	The PROGRESS_CALLBACK function must specify a single argument, *Percent*, which MESH_DECIMATE sets to an integer between 0 and 100, inclusive.
	
	The PROGRESS_CALLBACK function may specify an optional USERDATA keyword, which MESH_DECIMATE sets to the variable provided in the PROGRESS_USERDATA keyword.
	
	The following code show an example of a progress callback function:
	
	`FUNCTION myProgressCallback, percent,$`
	`   USERDATA = myStruct`
	
	`oProgressBar = myStruct.oProgressBar`
	
	`; This method updates the progress bar`
	`; graphic and returns TRUE if the user has`
	`; NOT pressed the cancel button.`
	`keepGoing = oProgressBar -> $`
	`   UpdateProgressValue(percent)`
	
	`RETURN, keepGoing`
	
	`END`

Keyword or item	Description

PROGRESS_METHOD	Set this keyword to a scalar string containing the name of a function method that MESH_DECIMATE calls at PROGRESS_PERCENT intervals as it decimates the mesh. If this keyword is set, then the PROGRESS_OBJECT keyword must be set to an object reference that is an instance of a class that defines the specified method.
	
	The PROGRESS_METHOD function method callback has the same specification as the callback described in the PROGRESS_CALLBACK keyword, except that it is defined as an object class method:
	
	`FUNCTION myClass::myProgressCallback, $`
	`   percent, USERDATA = myStruct`

PROGRESS_OBJECT	Set this keyword to an object reference that is an instance of a class that defines the method specified with the PROGRESS_METHOD keyword. If this keyword is set, then the PROGRESS_METHOD keyword must also be set.
PROGRESS_PERCENT	Set this keyword to a scalar in the range [1, 100] to specify the interval between invocations of the callback function. The default value is 5 and IDL silently clamps other values to the range [1, 100]. For example, a value of 5 (the default) specifies MESH_DECIMATE will call the callback function when the decimation is 0% complete, 5% complete, 10% complete, ..., 95% complete, and 100% complete.
PROGRESS_USERDATA	Set this property to any IDL variable that MESH_DECIMATE passes to the callback function in the callback function's USERDATA keyword parameter. If this keyword is specified, then the callback function must be able to accept keyword parameters.

MESSAGE

Keyword or item	Description
REISSUE_LAST	Set this keyword to reissue the last error issued by IDL. By using this keyword in conjunction with the CATCH procedure, your code can catch an error caused by called code, perform recovery actions, and issue the error normally. **Note -** If this keyword is specified, no plain arguments or other keywords may be specified. Combining the REISSUE_LAST keyword with arguments or other keywords will cause IDL to issue an error.

RESOLVE_ALL

Keyword or item	Description
CLASS	Set this keyword to a string or string array containing object class names. RESOLVE_ALL's rules for finding uncompiled functions and procedures are not able to find object definitions or methods, because those items are not known to IDL until the object classes are actually instantiated and the methods called. However, if the CLASS keyword is set, RESOLVE_ALL will ensure that the `*__define.pro` files for the specified classes and their superclasses are compiled and executed. RESOLVE_ALL then locates all methods for the specified classes and their superclasses, and makes sure they are also compiled.

SHMMAP

Keyword or item	Description
FILENAME	*The description of this keyword has been updated with the following text:* By default, files are mapped as shared, meaning that all processes that map the file will see any changes made. All changes are written back to the file by the operating system and become permanent. You must have write access to the file in order to map it as shared. To change the default behavior, set the PRIVATE keyword. When a file is mapped privately, changes made to the file are *not* written back to the file by the operating system, and are not visible to any other processes. You do not need write access to a file in order to map it privately — read access is sufficient.
PRIVATE	*The description of this keyword has been updated with the following text:* Mapping a file as shared requires that you have write access to the file, but a private mapping requires only read access. Use PRIVATE to map files for which you do not have write access, or when you want to ensure that the original file will not be altered by your process.

STRSPLIT

Keyword or item	Description
COUNT	Set this keyword to a named variable to receive the number of matched substrings returned by STRSPLIT. This value will be 0 in the case where either or both of the String and Pattern arguments is NULL. Otherwise, it is the number of elements in the result, equivalent to calling the N_ELEMENTS function.

WIDGET_BUTTON

Keyword or item	Description
PUSHBUTTON_EVENTS	Set this keyword to cause button events to be issued for the widget when the left mouse button is pressed and released, or when the spacebar is pressed and released. **Note -** This keyword has no effect on exclusive or non-exclusive buttons. When this keyword is *not* set, pressing and releasing either the left mouse button or the spacebar (if the button is in focus) generates a single button event, with the SELECT field set equal to 1. When this keyword *is* set: • Pressing the left mouse button generates a button event with the SELECT field set equal to 1. • Releasing the left mouse button generates a button event with the SELECT field set equal to 0. • Pressing the spacebar generates a button event with the SELECT field set equal to 1. • Releasing the spacebar generates a button event with the SELECT field set equal to 0. • Pressing and holding the spacebar generates a series of button events, with the value of the SELECT field alternating between 1 and 0. The rate at which events are generated is governed by the key-repeat settings of the operating system. **Note -** The spacebar only causes a button event if the button is in focus, which usually implies the button has been clicked on previously.

WIDGET_CONTROL

Keyword or item	Description
PUSHBUTTON_EVENTS	This keyword applies to widgets created with the WIDGET_BUTTON function. Set this keyword to a non-zero value to enable pushbutton events for the widget specified by *Widget_ID*. Set the keyword to 0 to disable pushbutton events for the specified widget.
REFRESH_PROPERTY	This keyword applies to widgets created with the WIDGET_PROPERTYSHEET function. Set this keyword to a property identifier or array of property identifiers to have just those properties synchronized with their values in the component(s). Recall that property identifiers are strings that uniquely determine a property. The keyword can also be set to a numeric value — non-zero values refresh all properties. The REFRESH_PROPERTY keyword also updates with respect to a property's sensitivity and visibility. When all properties need synchronizing, it is more efficient to use /REFRESH_PROPERTY than WIDGET_CONTROL’s SET_VALUE keyword to reload the property sheet.

WIDGET_INFO

Keyword or item	Description
COMPONENT	This keyword applies to widgets created with the WIDGET_PROPERTYSHEET function. Set this keyword to an object reference to indicate which object to query. This is most useful when the property sheet references multiple objects. If this keyword is not specified, the first (possibly only) object is queried.
PROPERTY_VALID	This keyword applies to widgets created with the WIDGET_PROPERTYSHEET function. Set this keyword to a string to determine if the string identifies a property. Valid identifiers return 1 and invalid strings return 0. Comparisons are case insensitive. Operations are performed on properties through unique identifiers. This operation is not required when processing a change event because the identifier returned in the event structure can be assumed to be correct.

Keyword or item	Description

PROPERTY_VALUE	This keyword applies to widgets created with the WIDGET_PROPERTYSHEET function. Retrieves the value of an identified property from a property sheet and returns it as a temporary IDL variable. Set this keyword to a string that is a valid property identifier to return the value of the specified property. This value can then be used to set the actual value of the property — the property sheet does not automatically do this. When there are multiple components, use the COMPONENT keyword to indicate which component should be queried. The match is case insensitive. An invalid identifier throws an error.
	
	This keyword is very often used in response to property sheet change events. This is because the property sheet does not change the underlying component; it only informs the widget program which of its own values have changed. The IDL programmer can use PROPERTY_VALUE to retrieve the user's desired value (as cached in the property sheet) and then apply it to the component. The following snippet of code handles property sheet change events:
	
	`PRO prop_event, e`
	
	`; get the value of e.component's`
	`; property identified by e.identifier`
	`value = WIDGET_INFO(e.id, $`
	`   COMPONENT = e.component, $`
	`   PROPERTY_VALUE = e.identifier )`
	
	`; set the component's property's value`
	`e.component -> SetPropertyByIdentifier, $`
	`   e.identifier, value`
	
	`END`

Keyword or item	Description
PUSHBUTTON_EVENTS	This keyword applies to widgets created with the WIDGET_BUTTON function. Set this keyword to return the pushbutton events status for the widget specified by *Widget_ID*. WIDGET_INFO returns 1 if pushbutton events are currently enabled for the widget, or 0 otherwise.

Routines Obsoleted in IDL 6.0

The following routines were present in IDL Version 5.6 but became obsolete in Version 6.0. These routines have been replaced with a new keyword to an existing routine or by a new routine that offers enhanced functionality. These obsoleted routines should not be used in new IDL code.

Routine	Replaced By
LIVE_CONTOUR	ICONTOUR
LIVE_CONTROL	iTools system
LIVE_DESTROY	iTools system
LIVE_EXPORT	iTools system
LIVE_IMAGE	IIMAGE
LIVE_INFO	iTools system
LIVE_LINE	iTools system
LIVE_LOAD	iTools system
LIVE_OPLOT	IPLOT
LIVE_PLOT	IPLOT
LIVE_PRINT	iTools system
LIVE_RECT	iTools system
LIVE_STYLE	iTools system
LIVE_SURFACE	ISURFACE
LIVE_TEXT	iTools system

Requirements for this Release

IDL 6.0 Requirements

Hardware Requirements for IDL 6.0

The following table describes the supported platforms and operating systems for IDL 6.0:

Platform	Vendor	Hardware	Operating System	Supported Versions
Windows	Microsoft	Intel x86 32-bit	Windows NT	[4.0], 2000, XP
Macintosh	Apple	G4 32-bit OS	Mac OS X	10.2.*x*†
UNIX†	Compaq	Alpha 64-bit*	Tru64 UNIX	5.1
	HP	PA-RISC 32-bit	HP-UX	11.0
	HP	PA-RISC 64-bit*	HP-UX	11.0
	IBM	RS/6000 32-bit	AIX	5.1
	IBM	RS/6000 64-bit*	AIX	5.1
	Intel	Intel x86 32-bit	Linux	Red Hat [7.1], 8, 9††
	SGI	Mips 32-bit	IRIX	6.5.1
	SGI	Mips 64-bit*	IRIX	6.5.1
	SUN	SPARC 32-bit	Solaris 2	[8], 9
	SUN	SPARC 64-bit*	Solaris 2	[8], 9

Table 1-1: Hardware Requirements for IDL 6.0.

On platforms with both 32-bit and 64-bit support, both versions are installed, and the 64-bit version is the default. The 32-bit version can be run by specifying the `-32` switch at the command line:

```
% idl -32
```

* The DXF file format and IDL DataMiner are not supported on 64-bit IDL platforms.

† For UNIX (including Mac OS X), the supported versions indicate that IDL was either built on the lowest version listed or tested on that version. You can install and run IDL on other versions that are binary compatible with those listed.

†† IDL 6.0 was built on the Linux 2.4 kernel with `glibc` 2.2 using Red Hat Linux 7.1. If your version of Linux is compatible with these, it is possible that you can install and run IDL on your version.

[] When multiple supported versions are listed, the bracketed version represents the operating system used to build IDL. Operating system versions that are binary compatible with the build version should run IDL 6.0 without problems, but only the versions listed in the table have been tested by RSI.

Software Requirements for IDL 6.0

The following table describes the software requirements for IDL 6.0:

Platform	Software Requirements
Windows	Internet Explorer 5.0 or higher.
Macintosh	MacOSX X11 which can be obtained at http://www.apple.com/macosx/x11.

Table 1-2: Software Requirements for IDL 6.0

ION 2.0 Requirements

Hardware Requirements for ION 2.0

The following table describes the supported platforms and operating systems for ION 2.0:

Platform	Vendor	Hardware	Operating System	Supported Versions
Windows	Microsoft	Intel x86 32-bit	Windows NT	4.0, 2000, XP
UNIX†	Intel	Intel x86 32-bit	Linux	Red Hat 7.1, 8, 9††
	SGI	Mips 32-bit	IRIX	6.5.1
	SUN	SPARC 32-bit	Solaris 2	8, 9

Table 1-3: Hardware Requirements for ION 2.0.

† For UNIX, the supported versions indicate that ION was either built on the lowest version listed or tested on that version. You can install and run ION on other versions that are binary compatible with those listed.

†† ION 2.0 was built on the Linux 2.4 kernel with `glibc` 2.2 using Red Hat Linux. If your version of Linux is compatible with these, it is possible that you can install and run ION 2.0 on your version.

Web Servers

In order to use ION, you must install an HTTP Web server. ION has been tested with the following Web server software:

- Apache Web Server version 2.0 or higher for Windows, Linux, and Solaris.
- Apache Web Server version 1.3.14 for IRIX. This version is included with the IRIX operating system.
- Microsoft Internet Information Server (IIS) version 5.0 for Windows 2000 Server and version 5.1 for Windows XP Professional.

If you do not already have Web server software, the IDL 6.0 CD-ROM contains the following Apache Web Server software:

- Windows — Version 2.0.45
- Linux — Version 2.0.43
- Solaris — Version 2.0.43
- IRIX — Version 1.3.14

Note

For more information on Apache software for your platform, see http://www.apache.org.

Web Browsers

ION 2.0 supports the HTTP 1.0 protocol. The following are provided as examples of popular Web browsers that support HTTP 1.0:

- Netscape Navigator versions 4.7 and 6.0.
- Microsoft Internet Explorer versions 5.5 and 6.0.

Browsers differ in their support of HTML features. As with any Web application, you should test your ION Script or Java application using Web browsers that anyone accessing your application is likely to be using.

Java Virtual Machines

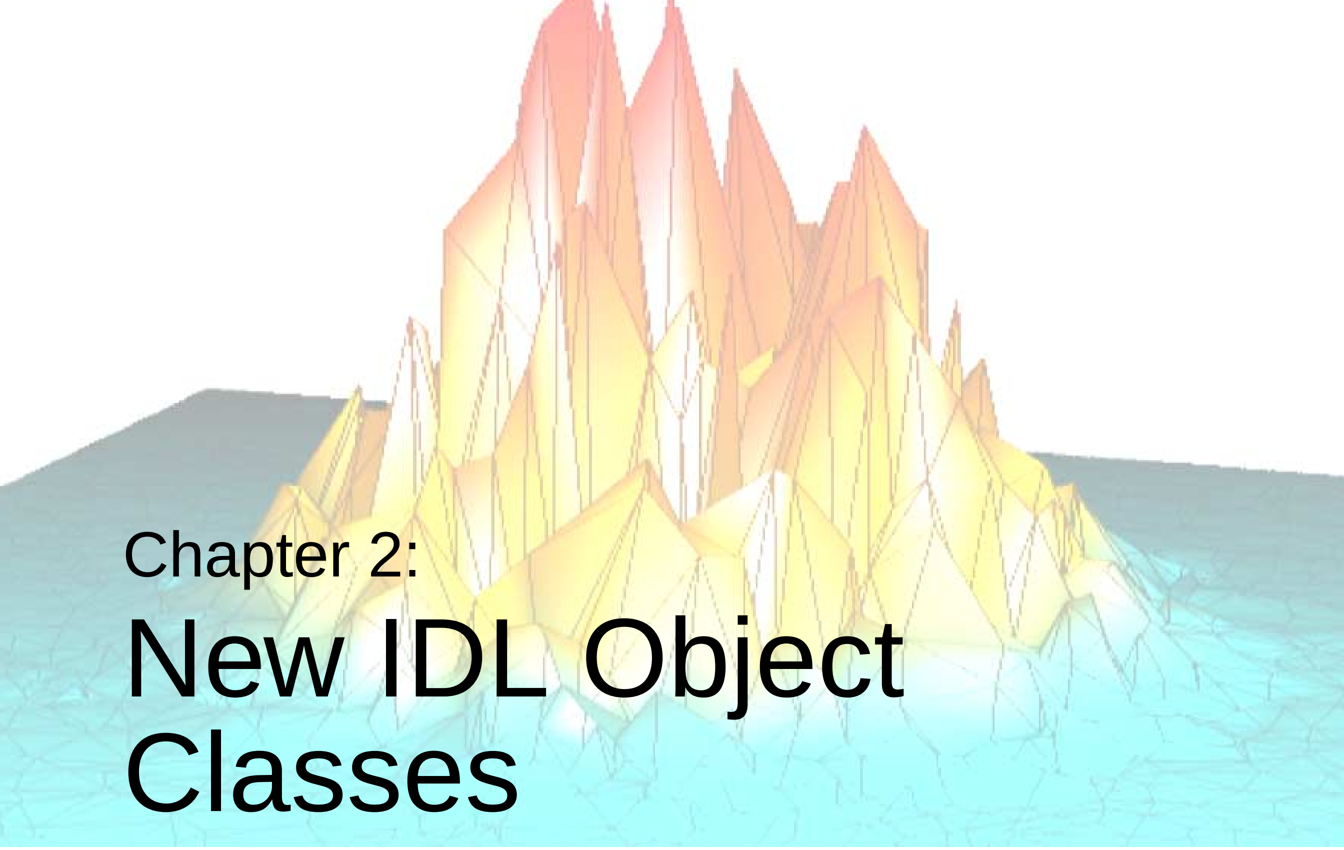
ION 2.0 supports the following Java Virtual Machines:

- Sun JVM 1.2, 1.3 and 1.4
- Microsoft JVM 5.x

The following are provided as examples of popular Web browsers that are shipped with the above JVMs:

- Netscape Navigator versions 4.7 and 6.0.
- Microsoft Internet Explorer versions 5.5 and 6.0.

Browsers differ in their support of features. As with any Web application, you should test your ION Java application using Web browsers that anyone accessing your application is likely to be using.



Chapter 2: New IDL Object Classes

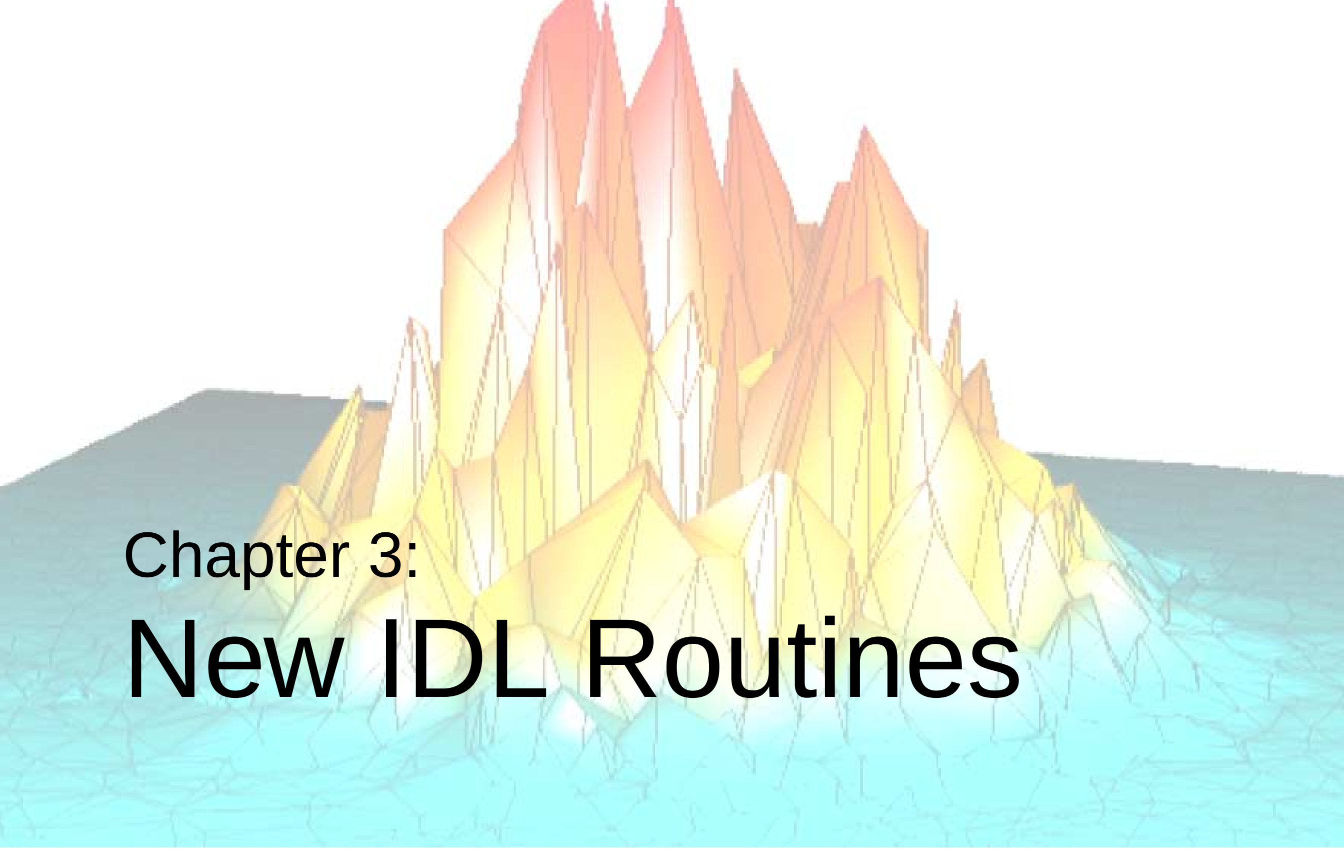
This chapter provides a list of new object classes introduced in IDL 6.0

List of New Object Classes

The following object classes are new in IDL 6.0:

- IDLitCommand
- IDLitCommandSet
- IDLitComponent
- IDLitContainer
- IDLitData
- IDLitDataContainer
- IDLitDataOperation
- IDLitIMessaging
- IDLitManipulator
- IDLitManipulatorContainer
- IDLitManipulatorManager
- IDLitManipulatorVisual
- IDLitOperation
- IDLitParameter
- IDLitParameterSet
- IDLitReader
- IDLitTool
- IDLitUI
- IDLitVisualization
- IDLitWindow
- IDLitWriter
- IDLjavaObject

These 22 new object classes contain a combined total of more than 250 methods. Because of the amount of new material, the detailed description on these classes and their properties and methods are provided only in Chapter 7, “iTools Object Classes” in the *IDL Reference Guide* manual and the IDL Online Help.



Chapter 3:

New IDL Routines

This chapter describes routines introduced in IDL version 6.0.

ARRAY_INDICES

The ARRAY_INDICES function converts one-dimensional subscripts of an array into corresponding multi-dimensional subscripts.

This routine is written in the IDL language. Its source code can be found in the file `array_indices.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = ARRAY_INDICES(*Array*, *Index*)

Return Value

If *Index* is a scalar, returns a vector containing *m* dimensional subscripts, where *m* is the number of dimensions of *Array*.

If *Index* is a vector containing *n* elements, returns an (*m* x *n*) array, with each row containing the multi-dimensional subscripts corresponding to that index.

Arguments

Array

An array of any type.

Index

A scalar or vector containing the one-dimensional subscripts to be converted.

Keywords

None.

Examples

Example 1

This example finds the location of the maximum value of a random 10 by 10 array:

```
seed = 111
array = RANDOMU(seed, 10, 10)
mx = MAX(array, location)
ind = ARRAY_INDICES(array, location)
PRINT, ind, array[ind[0],ind[1]], $
   FORMAT = '(%"Value at [%d, %d] is %f")'
```

IDL prints:

```
Value at [3, 6] is 0.973381
```

Example 2

This example routine locates the highest point in the example Maroon Bells data set and places a flag at that point.

Enter the following code in the IDL editor:

```
PRO ExARRAY_INDICES

; Import Maroon Bells data.
file = FILEPATH('surface.dat', $
   SUBDIRECTORY = ['examples', 'data'])
data = READ_BINARY(file, DATA_DIMS = [350, 450], $
   DATA_TYPE = 2)

; Display data.
ISURFACE, data

; Calculate the value and one-dimensional
; array location of the highest point.
maxValue = MAX(data, maxPoint)

; Using ARRAY_INDICES to convert the one-
; dimensional array location to a two-
; dimensional aray location.
maxLocation = ARRAY_INDICES(data, maxPoint)

; Print the results.
PRINT, 'Highest Point Location:  ', maxLocation
PRINT, 'Highest Point Value:  ', maxValue

; Create flag for the highest point.
```

```
x = maxLocation[0]
y = maxLocation[1]
z = maxValue
xFlag = [x, x, x + 50., x]
yFlag = [y, y, y + 50., y]
zFlag = [z, z + 1000., z + 750., z + 500.]

; Display flag at the highest point.
IPLOT, xFlag, yFlag, zFlag, /OVERPLOT

END
```

Save the code as `ExARRAY_INDICES.pro`, compile it and run it. The following figure displays the output of this example:

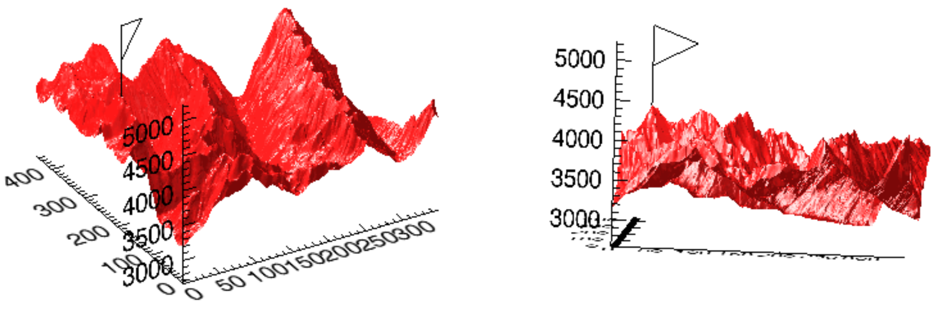


Figure 3-1: Maroon Bells Surface Plot with Flag at Highest Point Before Rotation (Left) and After Rotation (Right)

For a better view of the flag, use the Rotate tool to rotate the surface.

Version History

Introduced: 6.0

See Also

MAX, MIN, WHERE

FILE_BASENAME

The FILE_BASENAME function returns the *basename* of a *file path*. A file path is a string containing one or more segments consisting of names separated by directory delimiter characters (slash (`/`) under UNIX, or backslash (`\`) under Microsoft Windows). The basename is the final rightmost segment of the file path; it is usually a file, but can also be a directory name. See "Rules used by FILE_BASENAME" on page 128 for additional information.

Note

FILE_BASENAME operates on strings based strictly on their syntax. The *Path* argument need not refer to actual or existing files.

FILE_BASENAME is based on the standard UNIX `basename(1)` utility.

Note

To retrieve the leftmost portion of the file path (the *dirname*), use the FILE_DIRNAME function.

Syntax

Result = FILE_BASENAME(*Path* [, *RemoveSuffix*] [, /FOLD_CASE])

Return Value

A scalar string or string array containing the basename for each element of the *Path* argument.

Arguments

Path

A scalar string or string array containing paths for which the basename is desired.

Note

Under Microsoft Windows, the backslash (`\`) character is used to separate directories within a path. For compatibility with UNIX, and general convenience, the forward slash (`/`) character is also accepted as a directory separator in the *Path* argument.

RemoveSuffix

An optional scalar string or 1-element string array specifying a filename suffix to be removed from the end of the basename, if present.

Note

If the entire basename string matches the suffix, the suffix is *not* removed.

Keywords

FOLD_CASE

By default, FILE_BASENAME follows the case sensitivity policy of the underlying operating system when attempting to match a string specified by the *RemoveSuffix* argument. By default, matches are case sensitive on UNIX platforms, and case insensitive on Microsoft Windows platforms. The FOLD_CASE keyword is used to change this behavior. Set it to a non-zero value to cause FILE_BASENAME to do all string matching case insensitively. Explicitly set FOLD_CASE equal to zero to cause all string matching to be case sensitive.

Note

The value of the FOLD_CASE keyword is ignored if the *RemoveSuffix* argument is not present.

Rules used by FILE_BASENAME

FILE_BASENAME makes a copy of the input file path string, then modifies the copy according to the following rules:

- If *Path* is a NULL string, then FILE_BASENAME returns a NULL string.
- If *Path* consists entirely of directory delimiter characters, the result of FILE_BASENAME is a single directory delimiter character.
- If there are any trailing directory delimiter characters, they are removed.
- Under Microsoft Windows, remove any of the following, if present:
    - The drive letter and colon (for file paths of the form `c:\directory\file`).
    - The initial double-backslash and host name (for UNC file paths of the form `\\host\share\directory\file`).

- If any directory delimiter characters remain, all characters up to and including the last directory delimiter are removed.
- If the *RemoveSuffix* argument is present, is not identical to the characters remaining, and matches the *suffix* of the characters remaining, the suffix is removed. Otherwise, the *Result* is not modified by this step. The case sensitivity of the string comparison used in this step is controlled by the FOLD_CASE keyword.

Examples

The following command prints the basename of an IDL `.pro` file, removing the `.pro` suffix:

```
PRINT, FILE_BASENAME('/usr/local/rsi/idl/lib/dist.pro', '.pro')
```

IDL prints:

```
dist
```

Similarly, the following command prints the basenames of all `.pro` files in the `lib` subdirectory of the IDL distribution that begin with the letter “I,” performing a case insensitive match for the suffix:

```
PRINT, FILE_BASENAME(FILE_SEARCH(FILEPATH('lib')+'/i*.pro'),
   '.pro', /FOLD_CASE)
```

Version History

Introduced: 6.0

See Also

FILE_DIRNAME, PATH_SEP, STREGEX, STRMID, STRPOS, STRSPLIT

FILE_DIRNAME

The FILE_DIRNAME function returns the *dirname* of a *file path*. A file path is a string containing one or more segments consisting of names separated by directory delimiter characters (slash (`/`) under UNIX, or backslash (`\`) under Microsoft Windows). The dirname is all of the file path except for the final rightmost segment, which is usually a file name, but can also be a directory name. See "Rules use by FILE_DIRNAME" on page 131 for additional information.

Note

FILE_DIRNAME operates on strings based strictly on their syntax. The *Path* argument need not refer to actual or existing files.

FILE_DIRNAME is based on the standard Unix `dirname(1) utility`.

Note

To retrieve the rightmost portion of the file path (the *basename*), use the FILE_BASENAME function.

Syntax

Result = FILE_DIRNAME(*Path* [, /MARK_DIRECTORY])

Return Value

A scalar string or string array containing the dirname for each element of the *Path* argument.

Note

By default, the dirname does not include a final directory separator character; this behavior can be changed using the MARK_DIRECTORY keyword.

Note

On Windows platforms, the string returned by FILE_DIRNAME always uses the backslash (`\`) as the directory separator character, even if the slash (`/`) was used in the *Path* argument.

Arguments

Path

A scalar string or string array containing paths for which the dirname is desired.

Note

Under Microsoft Windows, the backslash (`\`) character is used to separate directories within a path. For compatibility with UNIX, and general convenience, the forward slash (`/`) character is also accepted as a directory separator in the *Path* argument. However, all *results* produced by FILE_DIRNAME on Windows platforms use the standard backslash for this purpose, regardless of the separator character used in the input *Path* argument.

Keywords

MARK_DIRECTORY

Set this keyword to include a directory separator character at the end of the returned directory name string. Including the directory character allows you to concatenate a file name to the end of the directory name string without having to supply the separator character manually. This is convenient for cross platform programming, as the separator characters differ between operating systems.

Rules use by FILE_DIRNAME

FILE_DIRNAME makes a copy of the input path string, and then modifies the copy according to the following rules:

- If *Path* is a NULL string, then FILE_DIRNAME returns a single dot (.) character, representing the current working directory of the IDL process.
- Under Microsoft Windows, a file path can start with either of the following:
  - A drive letter and a colon (for file paths of the form `c:\directory\file`).
  - An initial double-backslash and a host name (for UNC file paths of the form `\\host\share\directory\file`).

  If either of these are present in *Path*, they are considered to be part of the dirname, and are copied to the result *without interpretation by the remaining steps below.*

- If *Path* consists entirely of directory delimiter characters, the result of FILE_DIRNAME is a single directory delimiter character (prefixed by a Windows drive letter and colon or a UNC prefix, if necessary).
- All characters to the right of the rightmost directory delimiter character are removed.
- All trailing directory delimiter characters are removed.
- If the MARK_DIRECTORY keyword is set, a single directory delimiter character is appended to the end.

Examples

The following statements print the directory in which IDL locates the file `dist.pro` when it needs a definition for the DIST function. (DIST is part of the standard IDL user library, included with IDL):

```
temp = DIST(4) ; Ensure that DIST is compiled
PRINT, FILE_DIRNAME((ROUTINE_INFO('DIST', $
   /FUNCTION, /SCOURE)).path)
```

Depending on the platform and location where IDL is installed, IDL prints something like:

```
/usr/local/rsi/idl/lib
```

Version History

Introduced: 6.0

See Also

FILE_BASENAME, PATH_SEP, STREGEX, STRMID, STRPOS, STRSPLIT

ICONTOUR

The ICONTOUR procedure creates an iTool and associated user interface (UI) configured to display and manipulate contour data.

Note

If no arguments are specified, the ICONTOUR procedure creates an empty Contour tool.

This routine is written in the IDL language. Its source code can be found in the file `icontour.pro` in the `lib/itools` subdirectory of the IDL distribution.

Using Palettes

Contour colors can be specified in several ways. By default, all contour levels are black. The COLOR keyword can be used to change the color of all contour levels. For example, you can change contour levels to red by setting COLOR = [255, 0, 0]. Individual color levels can be specified when the iContour tool is in palette color mode, which allows a color table to be used. You can activate the palette color mode from the IDL Command Line by setting either of the RGB_TABLE or RGB_INDICES keywords, or from the iContour tool's property sheet by changing the **Use palette color** setting to `True`.

Note

If you are not in the palette color mode, the colors of individual levels may be modified in the contour level properties dialog. If you are in the palette color mode, the ability to edit individual colors in the contour level properties dialog is disabled. However, changing the **Use palette color** setting to `False` does not switch you back to previously set colors. It simply converts the colors referenced by indices to direct color values that can be individually modified. A common practice is to switch to palette color mode, select a palette, then change **Use palette color** to `False`. The colors of the palette are now loaded as individual contour colors that can each be edited in the contour level properties dialog.

If the iContour tool is in palette color mode, a colorbar can be inserted through the **Insert** menu. The colorbar displays a sample of the current palette associated with the contour display. The data values of the axis of the colorbar are based on the data range of the *Z* argument and the contour level values.

The minimum value of the colorbar axis represents the minimum of the data range. The maximum value of the axis is the greater of than the maximum of the data range and the highest contour level value.

Note

When IDL computes default contour levels, the highest contour level may be above the maximum value of the data.

Syntax

ICONTOUR[, *Z*[, *X*, *Y*]]

iTool Common Keywords: [, DIMENSIONS=[*x*, *y*]] [, IDENTIFIER=*variable*] [, LOCATION=[*x*, *y*]] [, NAME=*string*] [, OVERPLOT=*iToolID*] [, TITLE=*string*] [, VIEW_GRID=[*columns*, *rows*]] [, /VIEW_NEXT] [, VIEW_NUMBER=*integer*] [, {X | Y | Z}RANGE=[*min*, *max*]]

iTool Contour Keywords: [, RGB_INDICES=*vector of indices*] [, RGB_TABLE=*byte array of 256 by 3 or 3 by 256 elements*] [, ZVALUE=*value*]

Contour Object Keywords: [, AM_PM=*vector of two strings*] [, ANISOTROPY=[*x*, *y*, *z*]] [, C_COLOR=*color array*] [, C_FILL_PATTERN=*array of IDLgrPattern objects*] [, C_LABEL_INTERVAL=*vector*] [, C_LABEL_NOGAPS=*vector*] [, C_LABEL_OBJECTS=*array of object references*] [, C_LABEL_SHOW=*vector of integers*] [, C_LINESTYLE=*array of linestyles*] [, C_THICK=*float array*{each element 1.0 to 10.0}] [, C_USE_LABEL_COLOR=*vector of values*] [, C_USE_LABEL_ORIENTATION=*vector of values*] [, C_VALUE=*scalar or vector*] [, CLIP_PLANES=*array*] [, COLOR=*RGB vector*] [, DAYS_OF_WEEK=*vector of seven strings*] [, DEPTH_OFFSET=*value*] [, /DOWNHILL] [, /FILL] [, /HIDE] [, LABEL_FONT=*objref*] [, LABEL_FORMAT=*string*] [, LABEL_FRMTDATA=*value*] [, LABEL_UNITS=*string*] [, MAX_VALUE=*value*] [, MIN_VALUE=*value*] [, MONTHS=*vector of 12 values*] [, N_LEVELS=*value*] [, /PLANAR] [, SHADE_RANGE=[*min*, *max*]] [, SHADING={0 |1}] [, TICKINTERVAL=*value*] [, TICKLEN=*value*] [, USE_TEXT_ALIGNMENTS=*value*]

Axis Object Keywords: [, {X | Y | Z}GRIDSTYLE={0 | 1 | 2 | 3 | 4 | 5 | 6}]
[, {X | Y | Z}MAJOR=*integer*] [, {X | Y | Z}MINOR=*integer*]
[, {X | Y | Z}SUBTICKLEN=*ratio*] [, {X | Y | Z}TEXT_COLOR=*RGB vector*]
[, {X | Y | Z}TICKFONT_INDEX={0 | 1 | 2 | 3 | 4}]
[, {X | Y | Z}TICKFONT_SIZE=*integer*]
[, {X | Y | Z}TICKFONT_STYLE={0 | 1 | 2 | 3}]
[, {X | Y | Z}TICKFORMAT=*string or string array*]
[, {X | Y | Z}TICKINTERVAL=*value*] [, {X | Y | Z}TICKLAYOUT={0 | 1 | 2}]
[, {X | Y | Z}TICKLEN=*value*] [, {X | Y | Z}TICKNAME=*string array*]
[, {X | Y | Z}TICKUNITS=*string*] [, {X | Y | Z}TICKVALUES=*vector*]
[, {X | Y | Z}TITLE=*string*]

Arguments

X

A vector or two-dimensional array specifying the *x*-coordinates for the contour surface. If *X* is a vector, each element of *X* specifies the *x*-coordinate for a column of *Z* (e.g., X[0] specifies the *x*-coordinate for Z[0, *]). If X is a two-dimensional array, each element of X specifies the *x*-coordinate of the corresponding point in Z (i.e., X_{ij} specifies the *x*-coordinate for Z_{ij}).

Y

A vector or two-dimensional array specifying the *y*-coordinates for the contour surface. If *Y* is a vector, each element of *Y* specifies the *y*-coordinate for a row of Z (e.g., Y[0] specifies the *y*-coordinate for Z[*,0]). If *Y* is a two-dimensional array, each element of *Y* specifies the *y*-coordinate of the corresponding point in Z (Y_{ij} specifies the *y*-coordinate for Z_{ij}).

Z

A vector or two-dimensional array containing the values to be contoured. If the *X* and *Y* arguments are provided, the contour is plotted as a function of the (*x*, *y*) locations specified by their contents. Otherwise, the contour is generated as a function of the two-dimensional array index of each element of *Z*.

Keywords

Note

Because keywords to the ICONTOUR routine correspond to the names of registered properties of the iContour tool, the keyword names must be specified in full, without abbreviation.

AM_PM

Set this keyword to a vector of 2 strings indicating the names of the AM and PM strings when processing explicitly formatted dates (CAPA, CApA, and CapA format codes) with the LABEL_FORMAT keyword. See "Format Codes" in Chapter 10 of the *Building IDL Applications* manual for more information on format codes.

ANISOTROPY

Set this keyword equal to a three-element vector [*x, y, z*] that represents the multipliers to be applied to the internally computed correction factors along each axis that account for anisotropic geometry. Correcting for anisotropy is particularly important for the appropriate representations of downhill tickmarks.

By default, IDL will automatically compute correction factors for anisotropy based on the [XYZ] range of the contour geometry. If the geometry (as provided via the GEOMX, GEOMY, and GEOMZ keywords) falls within the range [*xmin*, *ymin*, *zmin*] to [*xmax*, *ymax*, *zmax*], then the default correction factors are computed as follows:

```
dx = xmax - xmin
dy = ymax - ymin
dz = zmax - zmin
; Get the maximum of the ranges:
maxRange = (dx > dy) > dz
IF (dx EQ 0) THEN xcorrection = 1.0 ELSE $
   xcorrection = maxRange / dx
IF (dy EQ 0) THEN ycorrection = 1.0 ELSE $
   ycorrection = maxRange / dy
IF (dz EQ 0) THEN zcorrection = 1.0 ELSE $
   zcorrection = maxRange / dz
```

This internally computed correction is then multiplied by the corresponding [*x, y, z*] values of the ANISOTROPY keyword. The default value for this keyword is [1,1,1]. IDL converts, maintains, and returns this data as double-precision floating-point.

C_COLOR

Set this keyword to a 3 by *N* array of RGB colors representing the colors to be applied at each contour level. If there are more contour levels than elements in this vector, the colors will be cyclically repeated. If C_COLOR is set to 0, all contour levels will be drawn in the color specified by the COLOR keyword (this is the default).

However, the C_COLOR keyword does not activate the palette color mode, which is recommended when working with contour levels and color. This mode can be activated with the RGB_INDICES and RGB_TABLE keywords. See "Using Palettes" on page 133 for more details.

C_FILL_PATTERN

Set this keyword to an array of IDLgrPattern objects representing the patterns to be applied at each contour level if the FILL keyword is non-zero. If there are more contour levels than fill patterns, the patterns will be cyclically repeated. If this keyword is set to 0, all contour levels are filled with a solid color (this is the default).

C_LABEL_INTERVAL

Set this keyword to a vector of values indicating the distance (measured parametrically relative to the length of each contour path) between labels for each contour level. If the number of contour levels exceeds the number of provided intervals, the C_LABEL_INTERVAL values will be repeated cyclically. The default is 0.4.

C_LABEL_NOGAPS

Set this keyword to a vector of values indicating whether gaps should be computed for the labels at the corresponding contour value. A zero value indicates that gaps will be computed for labels at that contour value; a non-zero value indicates that no gaps will be computed for labels at that contour value. If the number of contour levels exceeds the number of elements in this vector, the C_LABEL_NOGAPS values will be repeated cyclically. By default, gaps for the labels are computed for all levels (so that a contour line does not pass through the label).

C_LABEL_OBJECTS

Set this keyword to an array of object references to provide examples of labels to be drawn for each contour level. The objects specified via this keyword must inherit from one of the following classes:

- IDLgrSymbol
- IDLgrText

If a single object is provided, and it is an IDLgrText object, each of its strings will correspond to a contour level. If a vector of objects is used, any IDLgrText objects should have only a single string; each object will correspond to a contour level.

By default, with C_LABEL_OBJECTS set equal to a null object, IDL computes text labels that are the string representations of the corresponding contour level values.

Note

The objects specified via this keyword are used as descriptors only. The actual objects drawn as labels are generated by IDL.

The contour labels will have the same color as their contour level (see C_COLOR) unless the C_USE_LABEL_COLOR keyword is specified. The orientation of the label will be automatically computed unless the C_USE_LABEL_ORIENTATION keyword is specified. The horizontal and vertical alignment of any text labels will default to 0.5 (i.e., centered) unless the USE_TEXT_ALIGNMENTS keyword is specified.

Note

The object(s) set via this keyword will not be destroyed automatically when the contour is destroyed.

C_LABEL_SHOW

Set this keyword to a vector of integers. For each contour value, if the corresponding value in the C_LABEL_SHOW vector is non-zero, the contour line for that contour value will be labeled. If the number of contour levels exceeds the number of elements in this vector, the C_LABEL_SHOW values will be repeated cyclically. The default is 0 indicating that no contour levels will be labeled.

C_LINESTYLE

Set this keyword to an array of linestyles representing the linestyles to be applied at each contour level. The array may be either a vector of integers representing pre-defined linestyles, or an array of 2-element vectors representing a stippling pattern specification. If there are more contour levels than linestyles, the linestyles will be cyclically repeated. If this keyword is set to 0, all levels are drawn as solid lines (this is the default).

To use a pre-defined line style, set the C_LINESTYLE property equal to one of the following integer values:

- 0 = Solid line (the default)
- 1 = dotted
- 2 = dashed
- 3 = dash dot
- 4 = dash dot dot dot
- 5 = long dash
- 6 = no line drawn

To define your own stippling pattern, specify a two-element vector [*repeat*, *bitmask*], where *repeat* indicates the number of times consecutive runs of 1's or 0's in the *bitmask* should be repeated. (That is, if three consecutive 0's appear in the *bitmask* and the value of *repeat* is 2, then the line that is drawn will have six consecutive bits turned off.) The value of *repeat* must be in the range 1 ≤ *repeat* ≤ 255.

The *bitmask* indicates which pixels are drawn and which are not along the length of the line. *Bitmask* is most conveniently specified as a 16-bit hexadecimal value.

For example, `LINESTYLE = [2, 'F0F0'X]` describes a dashed line (8 bits on, 8 bits off, 8 bits on, 8 bits off).

C_THICK

Set this keyword to an array of line thicknesses representing the thickness to be applied at each contour level, where each element is a value between 1.0 and 10.0 points. If there are more contour levels than line thicknesses, the thicknesses will be cyclically repeated. If this keyword is set to 0, all contour levels are drawn with a line thickness of 1.0 points (this is the default).

C_USE_LABEL_COLOR

Set this keyword to a vector of values (0 or 1) to indicate whether the COLOR property value for each of the label objects (for the corresponding contour level) is to be used to draw that label. If the number of contour levels exceeds the number of elements in this vector, the C_USE_LABEL_COLOR values will be repeated cyclically. By default, this value is zero, indicating that the COLOR properties of the label objects will be ignored, and the C_COLOR property for the contour object will be used instead.

C_USE_LABEL_ORIENTATION

Set this keyword to a vector of values (0 or 1) to indicate whether the orientation for each of the label objects (for the corresponding contour level) is to be used when drawing the label. For text, the orientation of the object corresponds to the BASELINE and UPDIR property values; for a symbol, this refers to the default (un-rotated) orientation of the symbol. If the number of contour levels exceeds the number of elements in this vector, the C_USE_LABEL_ORIENTATION values will be repeated cyclically. By default, this value is zero, indicating that orientation of the label object(s) will be set to automatically computed values (to correspond to the direction of the contour paths).

C_VALUE

Set this keyword to a scalar value or a vector of values for which contour values are to be drawn. If this keyword is set to 0, contour levels will be evenly sampled across the range of the *Z* argument, using the value of the N_LEVELS keyword to determine the number of samples. IDL converts, maintains, and returns this data as double-precision floating-point.

CLIP_PLANES

Set this keyword to an array of dimensions [4, *N*] specifying the coefficients of the clipping planes to be applied to this object. The four coefficients for each clipping plane are of the form [A, B, C, D], where $Ax + By + Cz + D = 0$. Portions of this object that fall in the half space $Ax + By + Cz + D > 0$ will be clipped. By default, the value of this keyword is a scalar (-1) indicating that no clipping planes are to be applied.

Note

A window is only able to support a limited number of clipping planes. Some of these clipping planes may already be in use by the tool to support specific data display features. If the total number of clipping planes exceeds the limit, an informational message is displayed.

COLOR

Set this keyword to the color to be used to draw the contours. This color is specified as an RGB vector. The default is [0, 0, 0]. This value will be ignored if the C_COLOR keyword is set to a vector.

DAYS_OF_WEEK

Set this keyword to a vector of 7 strings to indicate the names to be used for the days of the week when processing explicitly formatted dates (CDWA, CDwA, and CdwA format codes) with the LABEL_FORMAT keyword. See "Format Codes" in Chapter 10 of the *Building IDL Applications* manual for more information on format codes.

DIMENSIONS

Set this keyword to a two-element vector of the form [width, height] to specify the dimensions of the drawing area of the specific tool in units specified by the UNITS keyword. If no value is provided, a default value of one half the screen size is used. The minimum width of the window correlates to the width of the menubar. The minimum window height is 100 pixels.

DOWNHILL

Set this keyword to indicate that downhill tick marks should be rendered as part of each contour level to indicate the downhill direction relative to the contour line.

FILL

Set this keyword to indicate that the contours should be filled. The default is to draw the contour levels as lines without filling. Filling contours may produce less than satisfactory results if your data contains NaNs, or if the contours are not closed.

HIDE

Set this keyword to a boolean value to indicate whether this object should be drawn:

- 0 = Draw graphic (the default)
- 1 = Do not draw graphic

IDENTIFIER

Set this keyword to a named variable that will contain the iToolID for the created tool. This value can then be used to reference this tool during overplotting operations or command-line-based tool management operations.

LABEL_FONT

Set this keyword to an instance of an IDLgrFont object to describe the default font to be used for contour labels. This font will be used for all text labels automatically generated by IDL (i.e., if C_LABEL_SHOW is set but the corresponding C_LABEL_OBJECTS text object is not provided), or for any text label objects provided via C_LABEL_OBJECTS that do not already have the font property set. The default value for this keyword is a NULL object reference, indicating that 12 pt. Helvetica will be used.

LABEL_FORMAT

Set this keyword to a string that represents a format string or the name of a function to be used to format the contour labels. If the string begins with an open parenthesis, it is treated as a standard format string. (Refer to the Format Codes in the IDL Reference Guide.) If the string does not begin with an open parenthesis, it is interpreted as the name of a callback function to be used to generate contour level labels.

The callback function is called with three parameters: *Axis*, *Index*, and *Value* and an optional DATA keyword, where:

- *Axis* is simply the value 2 to indicate that values along the Z axis are being formatted, which allows a single callback routine to be used for both axis labeling and contour labeling.
- *Index* is the contour level index (indices start at 0).
- *Value* is the data value of the current contour level.
- DATA is the optional keyword allowing any user-defined value specified through the LABEL_FRMTDATA keyword to ICONTOUR.

LABEL_FRMTDATA

Set this keyword to a value of any type. It will be passed via the DATA keyword to the user-supplied formatting function specified via the LABEL_FORMAT keyword, if any. By default, this value is 0, indicating that the DATA keyword will not be set (and furthermore, need not be supported by the user-supplied function).

LABEL_UNITS

Set this keyword to a string indicating the units to be used for default contour level labeling.

Valid unit strings include:

- "Numeric"
- "Years"
- "Months"
- "Days"
- "Hours"
- "Minutes"
- "Seconds"
- "Time" - Use this value to indicate that the contour levels correspond to time values; IDL will determine the appropriate label format based upon the range of values covered by the contour Z data.
- "" - The empty string is equivalent to the "Numeric" unit. This is the default.

If any of the time units are utilized, then the contour values are interpreted as Julian date/time values.

Note

The singular form of each of the time unit strings is also acceptable (for example, LEVEL_UNITS='Day' is equivalent to LEVEL_UNITS='Days').

LOCATION

Set this keyword to a two-element vector of the form [*x*, *y*] to specify the location of the upper left-hand corner of the tool relative to the display screen, in units specified by the UNITS keyword.

MAX_VALUE

Set this keyword to the maximum value to be plotted. Data values greater than this value are treated as missing data. The default is the maximum value of the input Z data. IDL converts, maintains, and returns this data as double-precision floating-point.

MONTHS

Set this keyword to a vector of 12 strings indicating the names to be used for the months when processing explicitly formatted dates (CMOA, CMoA, and CmoA format codes) with the C_LABEL_FORMAT keyword. See "Format Codes" in Chapter 10 of the *Building IDL Applications* manual for more information on format codes.

MIN_VALUE

Set this keyword to the minimum value to be plotted. Data values less than this value are treated as missing data. The default is the minimum value of the input Z data. IDL converts, maintains, and returns this data as double-precision floating-point.

NAME

Set this keyword to a string that specifies the name of this visualization.

N_LEVELS

Set this keyword to the number of contour levels to generate. This keyword is ignored if the C_VALUE 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 zero to indicate that IDL should compute a default number of levels based on the range of data values. This is the default.

OVERPLOT

Set this keyword to an iToolID to direct the graphical output of the particular tool to the tool specified by the provided iToolID.

Set this keyword to 1 (one) to place the graphical output for the command in the current tool. If no current tool exists, a new tool is created.

PLANAR

Set this keyword to indicate that the contoured data is to be projected onto a plane. Unlike the underlying IDLgrContour object, the default for ICONTOUR is planar (PLANAR = 1), which displays the contoured data in a plane. See the ZVALUE keyword to specify the Z value at which to display the planar Contour plot if it is displayed in a three dimensional data space.

RGB_INDICES

Set this keyword to a vector of indices into the color table to select colors to use for contour level colors. Setting the RGB_INDICES keyword activates the palette color mode, which allows colors from a specified color table to be used for the contour levels. The values set for RGB_INDICES are indices into the RGB_TABLE array of colors. If the number of colors selected using RGB_INDICES is less than the number of contour levels, the colors are repeated cyclically. If indices are not specified with the RGB_INDICES keyword, a default vector is constructed based on the values of the contour levels within the contour data range scaled to the byte range of RGB_TABLE.

See "Using Palettes" on page 133 for more details on the palette color mode.

RGB_TABLE

Set this keyword to either a 3 by 256 or 256 by 3 array containing color values to use for contour level colors. Setting the RGB_TABLE keyword activates the palette color mode, which allows colors from a specified color table to be used for the contour levels. The colors for each level are selected from RGB_TABLE using the RGB_INDICES vector. If indices are not specified with the RGB_INDICES keyword then a default vector is constructed based on the values of the contour levels within the contour data range scaled to the byte range of RGB_TABLE.

If the visualization is in palette color mode, but colors have not been specified with the RGB_TABLE keyword, the contour plot uses a default grayscale ramp.

See "Using Palettes" on page 133 for more details on the palette color mode.

SHADE_RANGE

Set this keyword to a two-element array that specifies the range of pixel values (color indices) to use for shading. The first element is the color index for the darkest pixel. The second element is the color index for the brightest pixel. This value is ignored when the contour is drawn to a graphics destination that uses the RGB color model.

SHADING

Set this keyword to an integer representing the type of shading to use:

- 0 = Flat (default): The color has a constant intensity for each face of the contour, based on the normal vector.
- 1 = Gouraud: The colors are interpolated between vertices, and then along scanlines from each of the edge intensities.

Gouraud shading may be slower than flat shading, but results in a smoother appearance.

TICKINTERVAL

Set this keyword equal to a number indicating the distance between downhill tickmarks, in data units. If TICKINTERVAL is not set, or if you explicitly set it to zero, IDL will compute the distance based on the geometry of the contour. IDL converts, maintains, and returns this data as double-precision floating-point.

TICKLEN

Set this keyword equal to a number indicating the length of the downhill tickmarks, in data units. If TICKLEN is not set, or if you explicitly set it to zero, IDL will compute the length based on the geometry of the contour. IDL converts, maintains, and returns this data as double-precision floating-point

TITLE

Set this keyword to a string to specify a title for the tool. The title is displayed in the title bar of the tool and is used for tool-related display purposes only – as the root of the hierarchy shown in the Tool Browser, for example.

USE_TEXT_ALIGNMENTS

Set this keyword to indicate that, for any IDLgrText labels (as specified via the C_LABEL_OBJECTS keyword), the ALIGNMENT and VERTICAL_ALIGNMENT property values for the given IDLgrText object(s) are to be used to draw the corresponding labels. By default, this value is zero, indicating that the ALIGNMENT and VERTICAL_ALIGNMENT properties of the label IDLgrText object(s) will be set to default values (0.5 for each, indicating centered labels).

VIEW_GRID

Set this keyword to a two-element vector of the form [columns, rows] to specify the view layout within the new tool. This keyword is only used if a new tool is being created (for example, if OVERPLOT, VIEW_NEXT, or VIEW_NUMBER are specified then VIEW_GRID is ignored).

VIEW_NEXT

Set this keyword to change the view selection to the next view following the currently-selected view before issuing any graphical commands. If the currently-selected view is the last one in the layout, then /VIEW_NEXT will cause the first view in the layout to become selected. This keyword is ignored if no current tool exists.

Note

The contents of the newly-selected view will be emptied unless /OVERPLOT is set.

VIEW_NUMBER

Set this keyword to change the currently-selected view to the view specified by the VIEW_NUMBER before issuing any graphical commands. The view number starts at 1, and corresponds to the position of the view within the graphics container (not necessarily the position on the screen). This keyword is ignored if no current tool exists.

Note

The contents of the newly-selected view will be emptied unless /OVERPLOT is set.

[XYZ]MAJOR

Set this keyword to an integer representing the number of major tick marks. The default is -1, specifying that IDL will compute the number of tickmarks. Setting MAJOR equal to zero suppresses major tickmarks entirely. ZMAJOR is ignored unless PLANAR is set to 0.

[XYZ]MINOR

Set this keyword to an integer representing the number of minor tick marks. The default is -1, specifying that IDL will compute the number of tickmarks. Setting MINOR equal to zero suppresses minor tickmarks entirely. ZMINOR is ignored unless PLANAR is set to 0.

[XYZ]RANGE

Set this keyword to the desired data range of the axis, a 2-element vector. The first element is the axis minimum, and the second is the maximum. ZRANGE is ignored unless PLANAR is set to 0.

[XYZ]SUBTICKLEN

Set this keyword to a floating-point scale ratio specifying the length of minor tick marks relative to the length of major tick marks. The default is 0.5, specifying that the minor tick mark is one-half the length of the major tick mark. ZSUBTICKLEN is ignored unless PLANAR is set to 0.

[XYZ]TEXT_COLOR

Set this keyword to an RGB value specifying the color for the axis text. The default value is [0, 0, 0] (black). ZTEXT_COLOR is ignored unless PLANAR is set to 0.

[XYZ]TICKFONT_INDEX

Set this keyword equal to one of the following integers, which represent the type of font to be used for the axis text:

- 0 = Helvetica
- 1 = Courier
- 2 = Times
- 3 = Symbol
- 4 = Hershey

ZTICKFONT_INDEX is ignored unless PLANAR is set to 0.

[XYZ]TICKFONT_SIZE

Set this keyword to an integer representing the point size of the font used for the axis text. The default is 12.0 points. ZTICKFONT_SIZE is ignored unless PLANAR is set to 0.

[XYZ]TICKFONT_STYLE

Set this keyword equal to one of the following integers, which represent the style of font to be used for the axis text:

- 0 = Normal
- 1 = Bold
- 2 = Italic
- 3 = Bold Italic

ZTICKFONT_STYLE is ignored unless PLANAR is set to 0.

[XYZ]TICKFORMAT

Set this keyword to a string, or an array of strings, in which each string represents a format string or the name of a function to be used to format the tick mark labels. If an array is provided, each string corresponds to a level of the axis. The TICKUNITS keyword determines the number of levels for an axis.

If the string begins with an open parenthesis, it is treated as a standard format string. See "Format Codes" in Chapter 10 of the *Building IDL Applications* manual.

If the string does not begin with an open parenthesis, it is interpreted as the name of a callback function to be used to generate tick mark labels.

If TICKUNITS are not specified:

- The callback function is called with three parameters: *Axis*, *Index*, and *Value*, where:
- *Axis* is the axis number: 0 for X axis, 1 for Y axis, 2 for Z axis
- *Index* is the tick mark index (indices start at 0)
- *Value* is the data value at the tick mark (a double-precision floating point value)

If TICKUNITS are specified:

The callback function is called with four parameters: *Axis*, *Index*, *Value*, and *Level*, where:

- *Axis*, *Index*, and *Value* are the same as described above.
- *Level* is the index of the axis level for the current tick value to be labeled. (Level indices start at 0.)

Used with the LABEL_DATE function, this property can easily create axes with date/time labels.

ZTICKFORMAT is ignored unless PLANAR is set to 0.

[XYZ]TICKINTERVAL

Set this keyword to a floating-point scalar indicating the interval between major tick marks for the first axis level. The default value is computed according to the axis [XYZ]RANGE and the number of major tick marks ([XYZ]MAJOR). The value of this keyword takes precedence over the value set for the [XYZ]MAJOR keyword.

For example, if TICKUNITS = ['S', 'H', 'D'], and TICKINTERVAL = 30, then the interval between major ticks for the first axis level will be 30 seconds.

ZTICKINTERVAL is ignored unless PLANAR is set to 0.

[XYZ]TICKLAYOUT

Set this keyword to integer scalar that indicates the tick layout style to be used to draw each level of the axis.

Valid values include:

- 0 = The axis line, major tick marks and tick labels are all included. Minor tick marks only appear on the first level of the axis. This is the default tick layout style.
- 1 = Only the labels for the major tick marks are drawn. The axis line, major tick marks, and minor tick marks are omitted.
- 2 = Each major tick interval is outlined by a box. The tick labels are positioned within that box (left-aligned). For the first axis level only, the major and minor tick marks will also be drawn.

ZTICKLAYOUT is ignored unless PLANAR is set to 0.

Note

For all tick layout styles, at least one tick label will appear on each level of the axis (even if no major tick marks fall along the axis line). If there are no major tick marks, the single tick label will be centered along the axis.

[XYZ]TICKLEN

Set this keyword to a floating-point value that specifies the length of each major tick mark, measured in data units. The recommended, and default, tick mark length is 0.2. IDL converts, maintains, and returns this data as double-precision floating-point. ZTICKLEN is ignored unless PLANAR is set to 0.

[XYZ]TICKNAME

Set this keyword to a string array of up to 30 elements that controls the annotation of each tick mark. ZTICKNAME is ignored unless PLANAR is set to 0.

[XYZ]TICKUNITS

Set this keyword to a string (or a vector of strings) indicating the units to be used for axis tick labeling. If more than one unit is provided, the axis will be drawn in multiple levels, one level per unit.

The order in which the strings appear in the vector determines the order in which the corresponding unit levels will be drawn. The first string corresponds to the first level (the level nearest to the primary axis line).

Valid unit strings include:

- "Numeric"
- "Years"
- "Months"
- "Days"
- "Hours"
- "Minutes"
- "Seconds"
- "Time" - Use this value to indicate that the tick values are time values; IDL will determine the appropriate time intervals and tick label formats based upon the range of values covered by the axis.
- ""- Use the empty string to indicate that no tick units are being explicitly set. This implies that a single axis level will be drawn using the "Numeric" unit. This is the default setting.

If any of the time units are utilized, then the tick values are interpreted as Julian date/time values. Note that the singular form of each of the time value strings is also acceptable (e.g, TICKUNITS = 'Day' is equivalent to TICKUNITS = 'Days').

ZTICKUNITS is ignored unless PLANAR is set to 0.

Note

Julian values must be in the range -1095 to 1827933925, which corresponds to calendar dates 1 Jan 4716 B.C.E. and 31 Dec 5000000hidd, respectively.

[XYZ]TICKVALUES

Set this keyword to a floating-point vector of data values representing the values at each tick mark. If TICKVALUES is set to 0, the default, IDL computes the tick values based on the axis range and the number of major ticks. IDL converts, maintains, and returns this data as double-precision floating-point. ZTICKVALUES is ignored unless PLANAR is set to 0.

[XYZ]TITLE

Set this keyword to a string representing the title of the specified axis. ZTITLE is ignored unless PLANAR is set to 0.

ZVALUE

For a planar contour plot, the height of the Z plane onto which the contour plot is projected.

Note

This keyword will not have any visual effect unless PLANAR is true and the plot is in a 3D dataspace, for example by selecting the **Surface** operation to add a surface plot to the dataspace along with the contour plot.

Examples

In the IDL Intelligent Tools system, data can be imported from the IDL Command Line (as described in Example 1), or data can be imported via the **File** menu in the iTool window (as described in Examples 2 and 3). For detailed information on importing data via the iTool file menu, refer to "Data Import Methods" in Chapter 2 of the *iTool User's Guide* manual.

Example 1

This example shows how to use the IDL Command Line to bring contour data into the iContour tool.

At the IDL Command Line, enter:

```
file = FILEPATH('convec.dat', SUBDIRECTORY = ['examples', 'data'])
data = READ_BINARY(file, DATA_DIMS = [248, 248])
ICONTOUR, data
```

Double-click on a contour to display the contour properties. Change the **Number of levels** setting to `20`, change **Use palette color** to `True`, and use the **Levels Color Table** setting to load the `EOS B` predefined color table through the **Load Predefined** button in the Palette Editor. Then, change the **Fill contours** setting to `True`.

The following figure displays the output of this example:

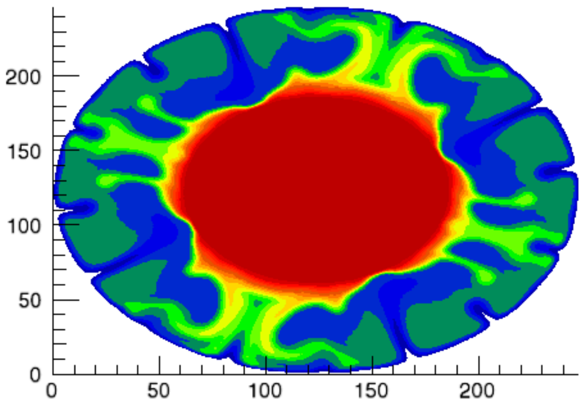


Figure 3-2: Earth Mantle Convection iContour Example

Example 2

This example shows how to use the iTool **File** → **Open** command to load DICOM data into the iContour tool.

At the IDL Command Line, enter:

```
ICONTOUR
```

Select **File** → **Open** to display the Open dialog, then browse to find `mr_brain.dcm` in the `examples/data` directory in the IDL distribution, and click **Open**.

Double-click on a contour to display the contour properties. Then, change **Use palette color** to `True` and the **Fill contours** setting to `True`.

Smooth the data by selecting **Operations** → **Filter** → **Smooth**.

The following figure displays the output of this example:

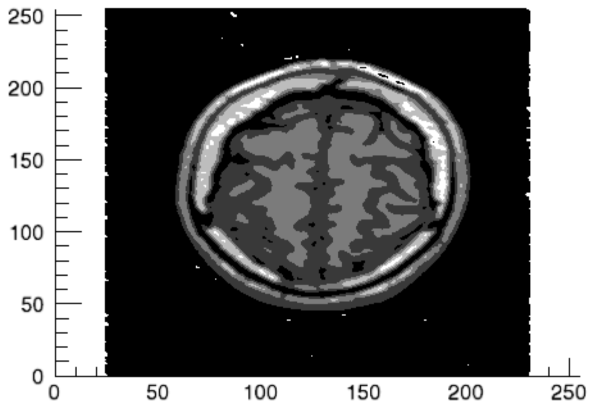


Figure 3-3: Smoothed Brain MRI iContour Example

Example 3

This example shows how to use the **File** → **Import** command to load binary data into the iContour tool.

At the IDL Command Line, enter:

```
ICONTOUR
```

Select **File** → **Import** to display the IDL Import Data wizard.

1. At Step 1, select **From a File** and click **Next**>>.
2. At Step 2, under **File Name:**, browse to find `idemosurf.dat` in the `examples/data` directory in the IDL distribution, and click **Next**>>.
3. At Step 3, select **Contour** and click **Finish**.

The Binary Template wizard is displayed. In the Binary Template, change **File's byte ordering** to `Little Endian`. Then, click **New Field**, and enter the following information in the New Field dialog:

- **Field Name:** `data` (or a name of your choosing)
- **Type:** `Float (32 bit)`
- **Number of Dimensions:** `2`

- **1st Dimension Size:** `200`
- **2nd Dimension Size:** `200`

Click **OK** to close the New Field dialog and the Binary Template dialog, and the contours are displayed.

Double-click on a contour to display the contour properties. Change the **Number of levels** setting to `10`, change **Use palette color** to `True`, and use the **Levels Color Table** setting to load the `Rainbow18` predefined color table through the **Load Predefined** button in the Palette Editor. Then, change the **Fill contours** setting to `True`.

Change the **Projection** setting from `Planar` to `Three-D`.

The following figure displays the output of this example:

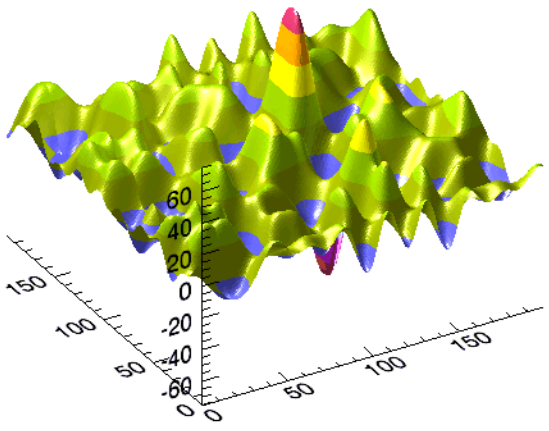


Figure 3-4: Filled Three-DImensional iContour Example

Version History

Introduced: 6.0

IDL_VALIDNAME

The IDL_VALIDNAME function determines whether a string may be used as a valid IDL variable name or structure tag name. Optionally, the routine can convert non-valid characters into underscores, returning a valid name string.

Syntax

Result = IDL_VALIDNAME(*String* [, /CONVERT_ALL] [, /CONVERT_SPACES])

Return Value

Returns the input string, optionally converting all spaces or non-alphanumeric characters to underscores. If the input string cannot be used as a valid variable or structure tag name, a null string is returned.

Arguments

String

A string representing the IDL variable or structure tag name to be checked.

Keywords

CONVERT_ALL

If this keyword is set, then *String* is converted into a valid IDL variable name using the following rules:

- All non-alphanumeric characters (except '_', '!' and '$') are converted to underscores
- If the first character of *String* is a number or a '$', then an underscore is prepended to the string
- If the first character of *String* is not a valid character ('_', '!', 'A'…'Z') then that character is converted to an underscore
- If *String* is an empty string or a reserved word (such as "AND") then an underscore is prepended to the string

Tip

The CONVERT_ALL keyword guarantees that a valid variable name is returned. It is useful in converting user-supplied strings into valid IDL variable names.

CONVERT_SPACES

If this keyword is set, then all spaces within *String* are converted to underscores. If *String* contains any other non-alphanumeric characters, then a null string is returned, indicating that the string cannot be used as a valid variable name.

Note

CONVERT_SPACES behaves the same as CREATE_STRUCT when checking structure tag names.

Examples

The following table provides IDL_VALIDNAME examples and their results.

Example	Result
`result = IDL_VALIDNAME('abc')`	`'abc'`
`result = IDL_VALIDNAME(' a b c ')`	`''`
`result = IDL_VALIDNAME(' a b c ', /CONVERT_SPACES)`	`'_a_b_c_'`
`result = IDL_VALIDNAME('$var')`	`''`
`result = IDL_VALIDNAME('$var', /CONVERT_ALL)`	`'_$VAR'`
`result = IDL_VALIDNAME('and')`	`''`
`result = IDL_VALIDNAME('and', /CONVERT_ALL)`	`'_AND'`

Table 3-1: IDL_VALIDNAME Examples

Version History

Introduced: 6.0

See Also

CREATE_STRUCT

IDLITSYS_CREATETOOL

The IDLITSYS_CREATETOOL function creates an instance of the specified tool registered within the IDL Intelligent Tools system.

This routine is written in the IDL language. Its source code can be found in the file `idlitsys_createtool.pro` in the `lib/itools` subdirectory of the IDL distribution.

Syntax

Result = IDLITSYS_CREATETOOL(*StrTool*[, INITIAL_DATA=*data*]
[, OVERPLOT=*iToolID*] [, PANEL_LOCATION={0 | 1 | 2 | 3}]
[, VIEW_GRID=*vector*] [, /VIEW_NEXT] [, VIEW_NUMBER=*number*]
[, VISUALIZATION_TYPE=*vistype*])

Return Value

Returns an iToolID that can be used to reference the created tool at a later time.

Arguments

StrTool

The name of a tool that has been registered with the iTools system via the ITREGISTER routine.

Keywords

Note
Additional keywords/properties associated with the target visualization at the command line are passed to the underlying system to be applied to the created tool and visualizations.

INITIAL_DATA

Set this keyword to the data objects that are used to create the initial visualizations in the created tool.

OVERPLOT

Set this keyword to the iToolID of the tool in which the visualization is to be created. This iToolID can be obtained during the creation of a previous tool or from the ITGETCURRENT routine.

PANEL_LOCATION

Set this keyword to an integer value to control where a user interface panel should be displayed. Possible values are:

0	position the panel above the iTool window
1	position the panel below the iTool window
2	position the panel to the left of the iTool window.
3	position the panel to the right of the iTool window (this is the default).

VIEW_GRID

Set this keyword to a two-element vector of the form [*columns*, *rows*] to specify the view layout within the new tool. This keyword is only used if a new tool is being created; it is ignored if OVERPLOT, VIEW_NEXT, or VIEW_NUMBER are specified.

VIEW_NEXT

Set this keyword to change the view selection to the next view following the currently-selected view before issuing any graphical commands. If the currently-selected view is the last one in the layout, then /VIEW_NEXT will cause the first view in the layout to become selected. This keyword is ignored if no current tool exists.

Note

The contents of the newly-selected view will be emptied unless /OVERPLOT is set.

VIEW_NUMBER

Set this keyword to change the currently-selected view to the view specified by the VIEW_NUMBER before issuing any graphical commands. The view number starts at 1, and corresponds to the position of the view within the graphics container (not necessarily the position on the screen). This keyword is ignored if no current tool exists.

Note
The contents of the newly-selected view will be emptied unless /OVERPLOT is set.

VISUALIZATION_TYPE

Set this keyword to a string containing the name of a registered visualization type that should be used to visualize any data specified by the INITIAL_DATA keyword. If this keyword is not specified, the iTool will select a visualization type based on the data type of the input data.

Examples

See Chapter 5, “Example: Simple iTool” in the *iTool Developer’s Guide* manual.

Version History

Introduced: 6.0

See Also

ITREGISTER, Chapter 5, “Creating an iTool Launch Routine” in the *iTool Developer’s Guide* manual.

IIMAGE

The IIMAGE procedure creates an iTool and associated user interface (UI) configured to display and manipulate image data.

Note
If no arguments are specified, the IIMAGE procedure creates an empty Image tool.

This routine is written in the IDL language. Its source code can be found in the file `iimage.pro` in the `lib/itools` subdirectory of the IDL distribution.

Syntax

IIMAGE[, *Image*[, *X*, *Y*]]

iTool Common Keywords: [, DIMENSIONS=[*x*, *y*]] [, IDENTIFIER=*variable*] [, LOCATION=[*x*, *y*]] [, NAME=*string*] [, OVERPLOT=*iToolID*] [, TITLE=*string*] [, VIEW_GRID=[*columns*, rows]] [, /VIEW_NEXT] [, VIEW_NUMBER=*integer*] [, {X | Y}RANGE=[*min*, *max*]]

iTool Image Keywords: [, ALPHA_CHANNEL=*2-D array*] [, BLUE_CHANNEL=*2-D array*] [, GREEN_CHANNEL=*2-D array*] [, IMAGE_DIMENSIONS=[*width*, *height*]] [, IMAGE_LOCATION=[*x*, *y*]] [, RED_CHANNEL=*2-D array*] [, RGB_TABLE=*array of 256 by 3 or 3 by 256 elements*]

Image Object Keywords: [, CHANNEL=*hexadecimal bitmask*] [, CLIP_PLANES=*array*] [, /HIDE] [, /INTERPOLATE] [, /ORDER]

Axis Object Keywords: [, {X | Y}GRIDSTYLE={0 | 1 | 2 | 3 | 4 | 5 | 6}] [, {X | Y}MAJOR=*integer*] [, {X | Y}MINOR=*integer*] [, {X | Y}SUBTICKLEN=*ratio*] [, {X | Y}TEXT_COLOR=*RGB vector*] [, {X | Y}TICKFONT_INDEX={0 | 1 | 2 | 3 | 4}] [, {X | Y}TICKFONT_SIZE=*integer*] [, {X | Y}TICKFONT_STYLE={0 | 1 | 2 | 3}] [, {X | Y}TICKFORMAT=*string or string array*] [, {X | Y}TICKINTERVAL=*value*] [, {X | Y}TICKLAYOUT={0 | 1 | 2}] [, {X | Y}TICKLEN=*value*] [, {X | Y}TICKNAME=*string array*] [, {X | Y}TICKUNITS=*string*] [, {X | Y}TICKVALUES=*vector*] [, {X | Y}TITLE=*string*]

Arguments

Image

Either a vector, a two-dimensional, or a three-dimensional array representing the sample values to be displayed as an image.

If *Image* is a vector:

- The *X* and *Y* arguments must also be present and contain the same number of elements. In this case, a dialog will be presented that offers the option of gridding the data to a regular grid (the results of which will be displayed as a color-indexed image).

If *Image* is a two-dimensional array:

- If either dimension is 3:

  Image represents an array of *x*, *y*, and *z* values (either $[[x_0, y_0, z_0], [x_1, y_1, z_1], ..., [x_n, y_n, z_n]]$ or $[[x_0, x_1, ..., x_n], [y_0, y_1, ..., y_n], [z_0, z_1, ..., z_n]]$ where *n* is the length of the other dimension). In this case, the *X* and *Y* arguments, if present, will be ignored. A dialog will be presented that allows the option of gridding the data to a regular grid (the results of which will be displayed as a color-indexed image, using the *z* values as the image data values).

- If neither dimension is 3:

  Image represents an array of sample values to be displayed as a color-indexed image. If *X* and *Y* are provided, the sample values are defined as a function of the corresponding (*x*, *y*) locations; otherwise, the sample values are implicitly treated as a function of the array indices of each element of *Image*.

If *Image* is a three-dimensional array:

- If one of the dimensions is 3:

  Image is a 3 x *n* x *m*, *n* x 3 x *m*, or *n* x *m* x 3 array representing the red, green, and blue channels of the image to be displayed.

- If one of the dimensions is 4:

  Image is a 4 x *n* x *m*, *n* x 4 x *m*, or *n* x *m* x 4 array representing the red, green, blue, and alpha channels of the image to be displayed.

X

Either a vector or a two-dimensional array representing the *x*-coordinates of the image grid.

If the *Image* argument is a vector:

- *X* must be a vector with the same number of elements as *Image*.

If the *Image* argument is a two-dimensional array (for which neither dimension is 3):

- If *X* is a vector:

  Each element of *X* specifies the *x*-coordinates for a column of *Image* (e.g., *X*[0] specifies the *x*-coordinate for *Image*[0, *]).

- If *X* is a two-dimensional array:

  Each element of *X* specifies the *x*-coordinate of the corresponding point in *Image* (X_{ij} specifies the *x*-coordinate of $Image_{ij}$).

Y

Either a vector or a two-dimensional array representing the *y*-coordinates of the image grid.

If the *Image* argument is a vector:

- *Y* must be a vector with the same number of elements.

If the *Image* argument is a two-dimensional array:

- If *Y* is a vector:

  Each element of *Y* specifies the *y*-coordinates for a column of *Image* (e.g., *Y*[0] specifies the *y*-coordinate for *Image*[*, 0]).

- If *Y* is a two-dimensional array:

  Each element of *Y* specifies the *y*-coordinate of the corresponding point in *Image* (Y_{ij} specifies the y-coordinate of $Image_{ij}$).

Keywords

Note

Because keywords to the IIMAGE routine correspond to the names of registered properties of the iImage tool, the keyword names must be specified in full, without abbreviation.

ALPHA_CHANNEL

Set this keyword to a two-dimensional array representing the alpha channel pixel values for the image to be displayed. This keyword is ignored if the *Image* argument is present, and is intended to be used in conjunction with some combination of the RED_CHANNEL, GREEN_CHANNEL, and BLUE_CHANNEL keywords.

BLUE_CHANNEL

Set this keyword to a two-dimensional array representing the blue channel pixel values for the image to be displayed. This keyword is ignored if the *Image* argument is present, and is intended to be used in conjunction with some combination of the RED_CHANNEL, GREEN_CHANNEL, and ALPHA_CHANNEL keywords.

CHANNEL

Set this keyword to a hexadecimal bitmask that defines which color channel(s) to draw. Each bit that is a 1 is drawn; each bit that is a 0 is not drawn. For example, 'ff0000'X represents a Blue channel write. The default is to draw all channels, and is represented by the hexadecimal value 'ffffff'X.

CLIP_PLANES

Set this keyword to an array of dimensions [4, *N*] specifying the coefficients of the clipping planes to be applied to this object. The four coefficients for each clipping plane are of the form [A, B, C, D], where Ax + By + Cz + D = 0. Portions of this object that fall in the half space Ax + By + Cz + D > 0 will be clipped. By default, the value of this keyword is a scalar (-1) indicating that no clipping planes are to be applied.

Note

A window is only able to support a limited number of clipping planes. Some of these clipping planes may already be in use by the tool to support specific data display features. If the total number of clipping planes exceeds the limit, an informational message is displayed.

DIMENSIONS

Set this keyword to a two-element vector of the form [width, height] to specify the dimensions of the drawing area of the specific tool in device units. The minimum width of the window correlates to the width of the menubar. The minimum window height is 100 pixels.

GREEN_CHANNEL

Set this keyword to a two-dimensional array representing the green channel pixel values for the image to be displayed. This keyword is ignored if the *Image* argument is present, and is intended to be used in conjunction with some combination of the RED_CHANNEL, BLUE_CHANNEL, and ALPHA_CHANNEL keywords.

HIDE

Set this keyword to a boolean value indicating whether this object should be drawn:

- 0 = Draw graphic (the default)
- 1 = Do not draw graphic

IDENTIFIER

Set this keyword to a named IDL variable that will contain the iToolID for the created tool. This value can then be used to reference this tool during overplotting operations or command-line-based tool management operations.

IMAGE_DIMENSIONS

Set this keyword to a 2-element vector, [*width*, *height*], to specify the image dimensions (in data units). By default, the dimensions match the pixel width of the image.

IMAGE_LOCATION

Set this keyword to a 2-element vector, [*x*, *y*], to specify the image location (in data units). By default, the location is [0, 0].

INTERPOLATE

Set this keyword to one (1) to display the iImage tool using bilinear interpolation. The default is to use nearest neighbor interpolation.

LOCATION

Set this keyword to a two-element vector of the form [*x*, *y*] to specify the location of the upper left-hand corner of the tool relative to the display screen, in device units.

NAME

Set this keyword to a string to specify the name for this particular tool. The name is used for tool-related display purposes only–as the root of the hierarchy shown in the Tool Browser, for example.

ORDER

Set this keyword to force the rows of the image data to be drawn from top to bottom. By default, image data is drawn from the bottom row up to the top row.

OVERPLOT

Set this keyword to an iToolID to direct the graphical output of the particular tool to the tool specified by the provided iToolID.

Set this keyword to 1 (one) to place the graphical output for the command in the current tool. If no current tool exists, a new tool is created.

RED_CHANNEL

Set this keyword to a two-dimensional array representing the red channel pixel values for the image to be displayed. This keyword is ignored if the *Image* argument is present, and is intended to be used in conjunction with some combination of the GREEN_CHANNEL, BLUE_CHANNEL, and ALPHA_CHANNEL keywords.

RGB_TABLE

Set this keyword to a 3 by 256 or 256 by 3 byte array of RGB color values. If no color tables are supplied, the tool will provide a default 256-entry linear grayscale ramp.

TITLE

Set this keyword to a string to specify a title for the tool. The title is displayed in the title bar of the tool.

VIEW_GRID

Set this keyword to a two-element vector of the form [*columns*, *rows*] to specify the view layout within the new tool. This keyword is only used if a new tool is being created (for example, if OVERPLOT, VIEW_NEXT, or VIEW_NUMBER are specified then VIEW_GRID is ignored).

VIEW_NEXT

Set this keyword to change the view selection to the next view following the currently-selected view before issuing any graphical commands. If the currently-selected view is the last one in the layout, then /VIEW_NEXT will cause the first view in the layout to become selected. This keyword is ignored if no current tool exists.

Note

The contents of the newly-selected view will be emptied unless /OVERPLOT is set.

VIEW_NUMBER

Set this keyword to change the currently-selected view to the view specified by the VIEW_NUMBER before issuing any graphical commands. The view number starts at 1, and corresponds to the position of the view within the graphics container (not necessarily the position on the screen). This keyword is ignored if no current tool exists.

Note

The contents of the newly-selected view will be emptied unless /OVERPLOT is set.

[XY]MAJOR

Set this keyword to an integer representing the number of major tick marks. The default is -1, specifying that IDL will compute the number of tickmarks. Setting MAJOR equal to zero suppresses major tickmarks entirely.

[XY]MINOR

Set this keyword to an integer representing the number of minor tick marks. The default is -1, specifying that IDL will compute the number of tickmarks. Setting MINOR equal to zero suppresses minor tickmarks entirely.

[XY]RANGE

Set this keyword to the desired data range of the axis, a 2-element vector. The first element is the axis minimum, and the second is the maximum.

[XY]SUBTICKLEN

Set this keyword to a floating-point scale ratio specifying the length of minor tick marks relative to the length of major tick marks. The default is 0.5, specifying that the minor tick mark is one-half the length of the major tick mark.

[XY]TEXT_COLOR

Set this keyword to an RGB value specifying the color for the axis text. The default value is [0, 0, 0] (black).

[XY]TICKFONT_INDEX

Set this keyword equal to one of the following integers, which represent the type of font to be used for the axis text:

- 0 = Helvetica
- 1 = Courier
- 2 = Times
- 3 = Symbol
- 4 = Hershey

[XY]TICKFONT_SIZE

Set this keyword to an integer representing the point size of the font used for the axis text. The default is 12.0 points.

[XY]TICKFONT_STYLE

Set this keyword equal to one of the following integers, which represent the style of font to be used for the axis text:

- 0 = Normal
- 1 = Bold
- 2 = Italic
- 3 = Bold Italic

[XY]TICKFORMAT

Set this keyword to a string, or an array of strings, in which each string represents a format string or the name of a function to be used to format the tick mark labels. If an array is provided, each string corresponds to a level of the axis. The TICKUNITS keyword determines the number of levels for an axis.

If the string begins with an open parenthesis, it is treated as a standard format string. See "Format Codes" in Chapter 10 of the *Building IDL Applications* manual.

If the string does not begin with an open parenthesis, it is interpreted as the name of a callback function to be used to generate tick mark labels.

If TICKUNITS are not specified:

- The callback function is called with three parameters: *Axis*, *Index*, and *Value*, where:
- *Axis* is the axis number: 0 for X axis, 1 for Y axis, 2 for Z axis
- *Index* is the tick mark index (indices start at 0)
- *Value* is the data value at the tick mark (a double-precision floating point value)

If TICKUNITS are specified:

The callback function is called with four parameters: *Axis*, *Index*, *Value*, and *Level*, where:

- *Axis*, *Index*, and *Value* are the same as described above.
- *Level* is the index of the axis level for the current tick value to be labeled. (Level indices start at 0.)

Used with the LABEL_DATE function, this property can easily create axes with date/time labels.

[XY]TICKINTERVAL

Set this keyword to a floating-point scalar indicating the interval between major tick marks for the first axis level. The default value is computed according to the axis [XYZ]RANGE and the number of major tick marks ([XYZ]MAJOR). The value of this keyword takes precedence over the value set for the [XYZ]MAJOR keyword.

For example, if TICKUNITS = ['S', 'H', 'D'], and TICKINTERVAL = 30, then the interval between major ticks for the first axis level will be 30 seconds.

[XY]TICKLAYOUT

Set this keyword to integer scalar that indicates the tick layout style to be used to draw each level of the axis.

Valid values include:

- 0 = The axis line, major tick marks and tick labels are all included. Minor tick marks only appear on the first level of the axis. This is the default tick layout style.
- 1 = Only the labels for the major tick marks are drawn. The axis line, major tick marks, and minor tick marks are omitted.
- 2 = Each major tick interval is outlined by a box. The tick labels are positioned within that box (left-aligned). For the first axis level only, the major and minor tick marks will also be drawn.

Note

For all tick layout styles, at least one tick label will appear on each level of the axis (even if no major tick marks fall along the axis line). If there are no major tick marks, the single tick label will be centered along the axis.

[XY]TICKLEN

Set this keyword to a floating-point value that specifies the length of each major tick mark, measured in data units. The recommended, and default, tick mark length is 0.2. IDL converts, maintains, and returns this data as double-precision floating-point.

[XY]TICKNAME

Set this keyword to a string array of up to 30 elements that controls the annotation of each tick mark.

[XY]TICKUNITS

Set this keyword to a string (or a vector of strings) indicating the units to be used for axis tick labeling. If more than one unit is provided, the axis will be drawn in multiple levels, one level per unit.

The order in which the strings appear in the vector determines the order in which the corresponding unit levels will be drawn. The first string corresponds to the first level (the level nearest to the primary axis line).

Valid unit strings include:

- "Numeric"
- "Years"
- "Months"
- "Days"
- "Hours"
- "Minutes"
- "Seconds"
- "Time" - Use this value to indicate that the tick values are time values; IDL will determine the appropriate time intervals and tick label formats based upon the range of values covered by the axis.
- ""- Use the empty string to indicate that no tick units are being explicitly set. This implies that a single axis level will be drawn using the "Numeric" unit. This is the default setting.

If any of the time units are utilized, then the tick values are interpreted as Julian date/time values. Note that the singular form of each of the time value strings is also acceptable (e.g, TICKUNITS = 'Day' is equivalent to TICKUNITS = 'Days').

Note

Julian values must be in the range -1095 to 1827933925, which corresponds to calendar dates 1 Jan 4716 B.C.E. and 31 Dec 5000000hidd, respectively.

[XY]TICKVALUES

Set this keyword to a floating-point vector of data values representing the values at each tick mark. If TICKVALUES is set to 0, the default, IDL computes the tick values based on the axis range and the number of major ticks. IDL converts, maintains, and returns this data as double-precision floating-point.

[XY]TITLE

Set this keyword to a string representing the title of the specified axis.

Examples

In the IDL Intelligent Tools system, data can be imported from the IDL Command Line (as described in Example 1), or data can be imported via the **File** menu in the iTool window (as described in Examples 2 and 3). For detailed information on importing data via the iTool file menu, refer to “Data Import Methods” in Chapter 2 of the *iTool User’s Guide* manual.

Example 1

This example shows how use the IDL Command Line to load data into the iImage tool.

At the IDL Command Line, enter:

```
file = FILEPATH('mineral.png', $
   SUBDIRECTORY = ['examples', 'data'])
data = READ_PNG(file)
IIMAGE, data, TITLE = 'Electron Image of Mineral Deposits'
```

Double-click the image to display image properties, and use the **Image Palette** setting to load the `Stern Special` predefined color table through the **Load Predefined** button in the Palette Editor.

Use the Text Annotation tool to insert a title at the top of the image. Select **Insert** → **Colorbars** to insert a color bar at the bottom of the image. Double-click on the colorbar to display its properties, and change the **Title** setting to `Stern Special`.

The following figure displays the output of this example:

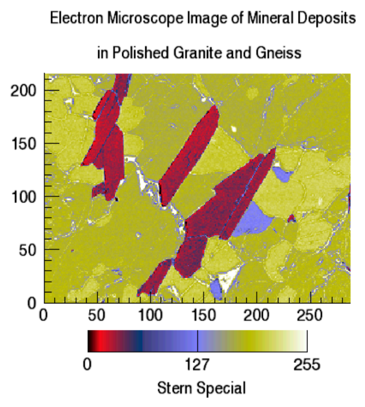


Figure 3-5: Mineral iImage Example with Sterns Color Table

Example 2

This example shows how to use the iTool **File** → **Open** command to load binary data into the iImage tool.

At the IDL Command Line, enter:

```
IIMAGE
```

Select **File** → **Open** to display the Open dialog, then browse to find `worldelv.dat` in the `examples/data` directory in the IDL distribution, and click **Open**.

In the Binary Template dialog, click **New Field**, and enter the following information in the New Field dialog:

- **Field Name:** `data` (or a name of your choosing)
- **Type:** `Byte (unsigned 8-bits)`
- **Number of Dimensions:** `2`
- **1st Dimension Size:** `360`
- **2nd Dimension Size:** `360`

Click **OK** to close the New Field dialog and the Binary Template dialog, and the image is displayed.

Note

For more information on using the Binary Template to import data, see “Using the BINARY_TEMPLATE Function” in Chapter 15 of the *Using IDL* manual.

Double-click the image to display image properties, and use the **Image Palette** setting to load the `STD GAMMA-II` predefined color table through the **Load Predefined** button in the Palette Editor.

The following figure displays the output of this example:

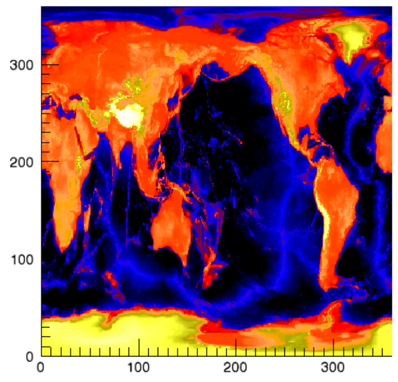


Figure 3-6: World Elevation iImage Example

Example 3

This example shows how to use the IDL Import Data Wizard to load image data into the iImage tool.

At the IDL Command Line, enter:

```
IIMAGE
```

Select **File** → **Import** to display the IDL Import Data wizard.

1. At Step 1, select **From a File** and click **Next**>>.
2. At Step 2, under **File Name:**, browse to find `n_vasinfecta.jpg` in the `examples/data` directory in the IDL distribution, and click **Next**>>.
3. At Step 3, select **Image** and click **Finish**.

Define the edges within the image by selecting **Operations** → **Filter** → **Sobel Filter**.

The following figure displays the output of this example:

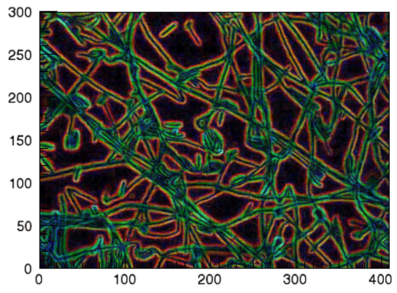


Figure 3-7: Sobel FIltered Neocosmospora Vasinfecta iImage Example

Version History

Introduced: 6.0

IPLOT

The IPLOT procedure creates an iTool and the associated user interface (UI) configured to display and manipulate plot data.

Note

If no arguments are specified, the IPLOT procedure creates an empty Plot tool.

This routine is written in the IDL language. Its source code can be found in the file `iplot.pro` in the `lib/itools` subdirectory of the IDL distribution.

Syntax

Cartesian

IPLOT, [*X*,] *Y*

or

IPLOT, *X*, *Y*, *Z*

Polar

IPLOT[, *R*], *Theta*, /POLAR

iTool Common Keywords: [, DIMENSIONS=[*x*, *y*]] [, IDENTIFIER=*variable*] [, LOCATION=[*x*, *y*]] [, NAME=*string*] [, OVERPLOT=*iToolID*] [, TITLE=*string*] [, VIEW_GRID=[*columns*, rows]] [, /VIEW_NEXT] [, VIEW_NUMBER=*integer*] [, {X | Y | Z}RANGE=[*min*, *max*]]

iTool Plot Keywords: [, ERRORBAR_COLOR=*RGB vector*] [, ERROR_CAPSIZE=*points*{0.0 to 1.0}] [, /FILL_BACKGROUND] [, FILL_COLOR=*RGB vector*] [, FILL_LEVEL=*value*] [, RGB_TABLE=*byte array of 256 by 3 or 3 by 256 elements*] [, /SCATTER] [, SYM_COLOR=*RGB vector*] [, SYM_INCREMENT=*integer*] [, SYM_INDEX=*integer*] [, SYM_SIZE=*points*{0.0 to 1.0}] [, SYM_THICK=*points*{1.0 to 10.0}] [, TRANSPARENCY=*percent*{0.0 to 100.0}] [, /USE_DEFAULT_COLOR] [, /XY_SHADOW] [, /{X | Y | Z}_ERRORBARS] [, /{X | Y | Z}_LOG] [, {X | Y | Z}ERROR=*vector* or *array*] [, /XZ_SHADOW] [, /YZ_SHADOW]

Plot Object Keywords: [, CLIP_PLANES=*array*] [, COLOR = *RGB vector*] [, /HIDE] [, /HISTOGRAM] [, LINESTYLE=*integer*] [, MAX_VALUE=*value*] [, MIN_VALUE=*value*] [, NSUM=*value*] [, /POLAR] [, THICK=*points*{1.0 to 10.0}] [, VERT_COLORS=*byte vector*]

Axis Object Keywords: [, {X | Y | Z}GRIDSTYLE={0 | 1 | 2 | 3 | 4 | 5 | 6}]
[, {X | Y | Z}MAJOR=*integer*] [, {X | Y | Z}MINOR=*integer*]
[, {X | Y | Z}SUBTICKLEN=*ratio*] [, {X | Y | Z}TEXT_COLOR=*RGB vector*]
[, {X | Y | Z}TICKFONT_INDEX={0 | 1 | 2 | 3 | 4}]
[, {X | Y | Z}TICKFONT_SIZE=*integer*]
[, {X | Y | Z}TICKFONT_STYLE={0 | 1 | 2 | 3}]
[, {X | Y | Z}TICKFORMAT=*string or string array*]
[, {X | Y | Z}TICKINTERVAL=*value*] [, {X | Y | Z}TICKLAYOUT={0 | 1 | 2}]
[, {X | Y | Z}TICKLEN=*value*] [, {X | Y | Z}TICKNAME=*string array*]
[, {X | Y | Z}TICKUNITS=*string*] [, {X | Y | Z}TICKVALUES=*vector*]
[, {X | Y | Z}TITLE=*string*]

Arguments

R

If the POLAR keyword is set, *R* is a vector representing the radius of the polar plot. If *R* is specified, *Theta* is plotted as a function of *R*. If *R* is not specified, *Theta* is plotted as a function of the vector index of *Theta*.

Theta

If the POLAR keyword is set, *Theta* is a vector representing the angle (in radians) of the polar plot.

X

A vector representing the *x*-coordinates of the plot.

Y

A vector or a two-dimensional array. If *Y* is:

- a vector, it represents the *y*-coordinates of the plot. If *X* is not specified, *Y* is plotted as a function of the vector index of *Y*. If *X* is specified, *Y* is plotted as a function of *X*.
- a 2-by-*n* array, *Y*[0, *] represents the *x*-coordinates and *Y*[1, *] represents the *y*-coordinates of the plot.
- a 3-by-*n* array, *Y*[0, *] represents the *x*-coordinates, *Y*[1, *] represents the *y*-coordinates, and *Y*[2, *] represents the *z*-coordinates of the plot.

Z

A vector representing the *z*-coordinates of the plot.

Keywords

Note

Because keywords to the IPLOT routine correspond to the names of registered properties of the iPlot tool, the keyword names must be specified in full, without abbreviation.

CLIP_PLANES

Set this keyword to an array of dimensions [4, *N*] specifying the coefficients of the clipping planes to be applied to this object. The four coefficients for each clipping plane are of the form [A, B, C, D], where A*x* + B*y* + C*z* + D = 0. Portions of this object that fall in the half space A*x* + B*y* + C*z* + D > 0 will be clipped. By default, the value of this keyword is a scalar (-1) indicating that no clipping planes are to be applied.

Note

A window is only able to support a limited number of clipping planes. Some of these clipping planes may already be in use by the tool to support specific data display features. If the total number of clipping planes exceeds the limit, an informational message is displayed.

COLOR

Set this keyword to an RGB value specifying the color to be used as the foreground color for this plot. The default is [0, 0, 0] (black).

DIMENSIONS

Set this keyword to a two-element vector of the form [width, height] to specify the dimensions of the drawing area of the specific tool in device units. The minimum width of the window correlates to the width of the menubar. The minimum window height is 100 pixels.

ERRORBAR_COLOR

Set this keyword to an RGB value specifying the color for the error bar. The default value is [0, 0, 0] (black).

ERRORBAR_CAPSIZE

Set this keyword to a floating-point value specifying the size of the error bar endcaps. This value ranges from 0 to 1.0, where a value of 1.0 results in an endcap that is 10% of the data range.

FILL_BACKGROUND (for 2D plots only)

Set this keyword to fill the area under the plot. This keyword is only available for two-dimensional plots. This keyword is ignored for three-dimensional plots.

FILL_COLOR (for 2D plots only)

Set this keyword to an RGB value specifying the color for the filled area. The default value is [255, 255, 255] (white). This keyword is only available for two-dimensional plots. This keyword is ignored for three-dimensional plots.

FILL_LEVEL (for 2D plots only)

Set this keyword to a floating-point value specifying the *y*-value for the lower boundary of the fill region. This keyword is only available for two-dimensional plots. This keyword is ignored for three-dimensional plots.

HIDE

Set this keyword to a boolean value indicating whether this object should be drawn:

- 0 = Draw graphic (the default)
- 1 = Do not draw graphic

HISTOGRAM (for 2D plots only)

Set this keyword to force only horizontal and vertical lines to be used to connect the plotted points. By default, the points are connected using a single straight line. This keyword is only available for two-dimensional plots. This keyword is ignored for three-dimensional plots.

IDENTIFIER

Set this keyword to a named IDL variable that will contain the iToolID for the created tool. This value can then be used to reference this tool during overplotting operations or command-line-based tool management operations.

LINESTYLE

Set this keyword to indicate the line style that should be used to draw the plot lines. The value can be either an integer value specifying a pre-defined line style, or a two-element vector specifying a stippling pattern.

To use a pre-defined line style, set the LINESTYLE keyword equal to one of the following integer values:

- 0 = Solid line (the default)
- 1 = dotted
- 2 = dashed
- 3 = dash dot
- 4 = dash dot dot dot
- 5 = long dash
- 6 = no line drawn

To define your own stippling pattern, specify a two-element vector [*repeat*, *bitmask*], where *repeat* indicates the number of times consecutive runs of 1's or 0's in the *bitmask* should be repeated. (That is, if three consecutive 0's appear in the *bitmask* and the value of *repeat* is 2, then the line that is drawn will have six consecutive bits turned off.) The value of *repeat* must be in the range 1 ≤ *repeat* ≤ 255.

The *bitmask* indicates which pixels are drawn and which are not along the length of the line. *Bitmask* is most conveniently specified as a 16-bit hexadecimal value.

For example, `LINESTYLE = [2, 'F0F0'X]` describes a dashed line (8 bits on, 8 bits off, 8 bits on, 8 bits off).

LOCATION

Set this keyword to a two-element vector of the form [*x*, *y*] to specify the location of the upper left-hand corner of the tool relative to the display screen, in device units.

MAX_VALUE

The maximum value to be plotted. If this keyword is present, data values greater than the value of MAX_VALUE are treated as missing data and are not plotted.

Note

The IEEE floating-point value NaN is also treated as missing data. IDL converts, maintains, and returns this data as double-precision floating-point.

MIN_VALUE

The minimum value to be plotted. If this keyword is present, data values less than the value of MIN_VALUE are treated as missing data and are not plotted.

Note

The IEEE floating-point value NaN is also treated as missing data. IDL converts, maintains, and returns this data as double-precision floating-point.

NAME

Set this keyword to a string to specify the name for this visualization.

NSUM

Set this keyword to the number of data points to average when plotting. If NSUM is larger than 1, every group of NSUM points is averaged to produce one plotted point. If there are M data points, then M/NSUM points are plotted.

OVERPLOT

Set this keyword to an iToolID to direct the graphical output of the particular tool to the tool specified by the provided iToolID.

Set this keyword to 1 (one) to place the graphical output for the command in the current tool. If no current tool exists, a new tool is created.

POLAR

Set this keyword to display the plot as a polar plot. If this keyword is set, the arguments will be interpreted as *R* and *Theta* or simply *Theta* for a single argument. If *R* is not supplied the plot will use a vector of indices for the *R* argument.

RGB_TABLE

Set this keyword to either a 3 by 256 or 256 by 3 byte array containing color values to use for vertex colors. If the values supplied are not of type byte, they are scaled to the byte range using BYTSCL. Use the VERT_COLORS keyword to specify indices that select colors from the values specified with RGB_TABLE.

SCATTER

Set this keyword to generate a scatter plot. This action is equivalent to setting LINESTYLE = 6 (no line) and SYM_INDEX = 3 (Period symbol).

SYM_COLOR

Set this keyword to an RGB value specifying the color for the plot symbol.

Note

This color is applied to the symbol only if the USE_DEFAULT_COLOR property is set.

SYM_INCREMENT

Set this keyword to an integer value specifying the number of vertices to increment between symbol instances. The default value is 1, which places a symbol on every vertex.

SYM_INDEX

Set this keyword to one of the following scalar-represented internal default symbols:

- 0 = No symbol
- 1 = Plus sign, ‘+’ (default)
- 2 = Asterisk
- 3 = Period (Dot)
- 4 = Diamond
- 5 = Triangle
- 6 = Square
- 7 = X
- 8 = Arrow Head

SYM_SIZE

Set this keyword to a floating-point value from 0.0 to 1.0 specifying the size of the plot symbol. A value of 1.0 results in an symbol that is 10% of the width/height of the plot.

SYM_THICK

Set this keyword to floating-point value from 1 to 10 points specifying the thickness of the plot symbol.

THICK

Set this keyword to a value between 1.0 and 10.0, specifying the line thickness to be used to draw the plotted lines, in points. The default is 1.0 points.

TITLE

Set this keyword to a string to specify a title for the tool. The title is displayed in the title bar of the tool and is used for tool-related display purposes only–as the root of the hierarchy shown in the Tool Browser, for example.

TRANSPARENCY

Set this keyword to floating-point value specifying the transparency of the filled area. Valid values range from 0.0 to 100.0. The default value is 0.0 (opaque).

USE_DEFAULT_COLOR

Set this keyword to have the color of the symbols match the plot color. If this keyword is set to 0 (USE_DEFAULT_COLOR = 0), the color specified by the SYM_COLOR keyword is used for symbols instead of matching the color of the plot.

VERT_COLORS

Set this keyword to a vector of indices into the color table to select colors to use for each vertex (plot data point). Alternately, set this keyword to a 3 by *N* byte array containing color values to use for each vertex. If the values supplied are not of type byte, they are scaled to the byte range using BYTSCL. If indices are supplied but no colors are provided with the RGB_TABLE keyword, then a default grayscale ramp is used. If a 3 by *N* array of colors is provided, the colors are used directly and the color values provided with RGB_TABLE are ignored. If the number of indices or colors specified is less than the number of vertices, the colors are repeated cyclically.

VIEW_GRID

Set this keyword to a two-element vector of the form [*columns*, *rows*] to specify the view layout within the new tool. This keyword is only used if a new tool is being created (for example, if OVERPLOT, VIEW_NEXT, or VIEW_NUMBER are specified then VIEW_GRID is ignored).

VIEW_NEXT

Set this keyword to change the view selection to the next view following the currently-selected view before issuing any graphical commands. If the currently-selected view is the last one in the layout, then /VIEW_NEXT will cause the first view in the layout to become selected. This keyword is ignored if no current tool exists.

Note

The contents of the newly-selected view will be emptied unless /OVERPLOT is set.

VIEW_NUMBER

Set this keyword to change the currently-selected view to the view specified by the VIEW_NUMBER before issuing any graphical commands. The view number starts at 1, and corresponds to the position of the view within the graphics container (not necessarily the position on the screen). This keyword is ignored if no current tool exists.

Note

The contents of the newly-selected view will be emptied unless /OVERPLOT is set.

XY_SHADOW (for 3D plots only)

Set this keyword to display a shadow of the plot in a three-dimensional plot. The shadow lies in the XY plane at the minimum value of the data space range of the *z*-axis. This keyword has no effect for two-dimensional plots.

[XYZ]_ERRORBARS

Set this keyword to show error bars. The Z_ERRORBARS keyword is for three-dimensional plots only.

[XYZ]_LOG

Set this keyword to specify a logarithmic axis. The minimum value of the axis range must be greater than zero. The Z_LOG keyword is for three-dimensional plots only.

[XYZ]ERROR

Set this keyword to either a vector or a 2 by *N* array of error values to be displayed as error bars for the [XYZ] dimension of the plot. The length of this array must be equal in length to the number of vertices of the plot or it will be ignored. If this keyword is set to a vector, the value will be applied as both a negative and positive error and the error bar will be symmetric about the plot vertex. If this keyword is set to a 2 by *N* array the [0, *] values define the negative error and the [1, *] values define the positive error, allowing asymmetric error bars. The ZERROR keyword is for three-dimensional plots only.

[XYZ]MAJOR

Set this keyword to an integer representing the number of major tick marks. The default is -1, specifying that IDL will compute the number of tickmarks. Setting MAJOR equal to zero suppresses major tickmarks entirely. ZMAJOR is for three-dimensional plots only.

[XYZ]MINOR

Set this keyword to an integer representing the number of minor tick marks. The default is -1, specifying that IDL will compute the number of tickmarks. Setting MINOR equal to zero suppresses minor tickmarks entirely. ZMINOR is for three-dimensional plots only.

[XYZ]RANGE

Set this keyword to the desired data range of the axis, a 2-element vector. The first element is the axis minimum, and the second is the maximum. ZRANGE is for three-dimensional plots only.

[XYZ]SUBTICKLEN

Set this keyword to a floating-point scale ratio specifying the length of minor tick marks relative to the length of major tick marks. The default is 0.5, specifying that the minor tick mark is one-half the length of the major tick mark. ZSUBTICKLEN is for three-dimensional plots only.

[XYZ]TEXT_COLOR

Set this keyword to an RGB value specifying the color for the axis text. The default value is [0, 0, 0] (black). ZTEXT_COLOR is for three-dimensional plots only.

[XYZ]TICKFONT_INDEX

Set this keyword equal to one of the following integers, which represent the type of font to be used for the axis text:

- 0 = Helvetica
- 1 = Courier
- 2 = Times
- 3 = Symbol
- 4 = Hershey

ZTICKFONT_INDEX is for three-dimensional plots only.

[XYZ]TICKFONT_SIZE

Set this keyword to an integer representing the point size of the font used for the axis text. The default is 12.0 points. ZTICKFONT_SIZE is for three-dimensional plots only.

[XYZ]TICKFONT_STYLE

Set this keyword equal to one of the following integers, which represent the style of font to be used for the axis text:

- 0 = Normal
- 1 = Bold
- 2 = Italic
- 3 = Bold Italic

ZTICKFONT_STYLE is for three-dimensional plots only.

[XYZ]TICKFORMAT

Set this keyword to a string, or an array of strings, in which each string represents a format string or the name of a function to be used to format the tick mark labels. If an array is provided, each string corresponds to a level of the axis. The TICKUNITS keyword determines the number of levels for an axis.

If the string begins with an open parenthesis, it is treated as a standard format string. See “Format Codes” in Chapter 10 of the *Building IDL Applications* manual.

If the string does not begin with an open parenthesis, it is interpreted as the name of a callback function to be used to generate tick mark labels.

If TICKUNITS are not specified:

- The callback function is called with three parameters: *Axis*, *Index*, and *Value*, where:
- *Axis* is the axis number: 0 for X axis, 1 for Y axis, 2 for Z axis
- *Index* is the tick mark index (indices start at 0)
- *Value* is the data value at the tick mark (a double-precision floating point value)

If TICKUNITS are specified:

The callback function is called with four parameters: *Axis*, *Index*, *Value*, and *Level*, where:

- *Axis*, *Index*, and *Value* are the same as described above.
- *Level* is the index of the axis level for the current tick value to be labeled. (Level indices start at 0.)

Used with the LABEL_DATE function, this property can easily create axes with date/time labels.

ZTICKFORMAT is for three-dimensional plots only.

[XYZ]TICKINTERVAL

Set this keyword to a floating-point scalar indicating the interval between major tick marks for the first axis level. The default value is computed according to the axis [XYZ]RANGE and the number of major tick marks ([XYZ]MAJOR). The value of this keyword takes precedence over the value set for the [XYZ]MAJOR keyword.

For example, if TICKUNITS = ['S', 'H', 'D'], and TICKINTERVAL = 30, then the interval between major ticks for the first axis level will be 30 seconds.

ZTICKINTERVAL is for three-dimensional plots only.

[XYZ]TICKLAYOUT

Set this keyword to integer scalar that indicates the tick layout style to be used to draw each level of the axis.

Valid values include:

- 0 = The axis line, major tick marks and tick labels are all included. Minor tick marks only appear on the first level of the axis. This is the default tick layout style.
- 1 = Only the labels for the major tick marks are drawn. The axis line, major tick marks, and minor tick marks are omitted.
- 2 = Each major tick interval is outlined by a box. The tick labels are positioned within that box (left-aligned). For the first axis level only, the major and minor tick marks will also be drawn.

ZTICKLAYOUT is for three-dimensional plots only.

Note

For all tick layout styles, at least one tick label will appear on each level of the axis (even if no major tick marks fall along the axis line). If there are no major tick marks, the single tick label will be centered along the axis.

[XYZ]TICKLEN

Set this keyword to a floating-point value that specifies the length of each major tick mark, measured in data units. The recommended, and default, tick mark length is 0.2. IDL converts, maintains, and returns this data as double-precision floating-point. ZTICKLEN is for three-dimensional plots only.

[XYZ]TICKNAME

Set this keyword to a string array of up to 30 elements that controls the annotation of each tick mark. ZTICKNAME is for three-dimensional plots only.

[XYZ]TICKUNITS

Set this keyword to a string (or a vector of strings) indicating the units to be used for axis tick labeling. If more than one unit is provided, the axis will be drawn in multiple levels, one level per unit.

The order in which the strings appear in the vector determines the order in which the corresponding unit levels will be drawn. The first string corresponds to the first level (the level nearest to the primary axis line).

Valid unit strings include:

- "Numeric"
- "Years"
- "Months"
- "Days"
- "Hours"
- "Minutes"
- "Seconds"
- "Time" - Use this value to indicate that the tick values are time values; IDL will determine the appropriate time intervals and tick label formats based upon the range of values covered by the axis.
- ""- Use the empty string to indicate that no tick units are being explicitly set. This implies that a single axis level will be drawn using the "Numeric" unit. This is the default setting.

If any of the time units are utilized, then the tick values are interpreted as Julian date/time values. Note that the singular form of each of the time value strings is also acceptable (e.g, TICKUNITS = 'Day' is equivalent to TICKUNITS = 'Days').

ZTICKUNITS is for three-dimensional plots only.

Note

Julian values must be in the range -1095 to 1827933925, which corresponds to calendar dates 1 Jan 4716 B.C.E. and 31 Dec 5000000hidd, respectively.

[XYZ]TICKVALUES

Set this keyword to a floating-point vector of data values representing the values at each tick mark. If TICKVALUES is set to 0, the default, IDL computes the tick values based on the axis range and the number of major ticks. IDL converts, maintains, and returns this data as double-precision floating-point. ZTICKVALUES is for three-dimensional plots only.

[XYZ]TITLE

Set this keyword to a string representing the title of the specified axis. ZTITLE is for three-dimensional plots only.

XZ_SHADOW (for 3D plots only)

Set this keyword to display a shadow of the plot in a three-dimensional plot. The shadow lies in the XZ plane at the minimum value of the data space range of the *y*-axis. This keyword has no effect for two-dimensional plots.

YZ_SHADOW (for 3D plots only)

Set this keyword to display a shadow of the plot in a three-dimensional plot. The shadow lies in the YZ plane at the minimum value of the data space range of the *x*-axis. This keyword has no effect for two-dimensional plots.

Examples

In the IDL Intelligent Tools system, data can be imported from the IDL Command Line (as described in Example 1), or data can be imported via the File menu in the iTool window (as described in Examples 2 and 3). For detailed information on importing data via the iTool file menu, refer to "Data Import Methods" in Chapter 2 of the *iTool User's Guide* manual.

Example 1

This example shows how to use the IDL Command Line to load data and variables into the iPlot tool.

At the IDL Command Line, enter:

```
file = FILEPATH('elnino.dat', SUBDIRECTORY = ['examples','data'])
data = READ_BINARY(file, DATA_TYPE = 4, DATA_DIMS = [500, 1], $
   ENDIAN = 'little')
time = DINDGEN(500)*0.25d + 1871
IPLOT, time, data, TITLE = 'El Nino', COLOR = [255, 128, 0]
```

Place a title on the time axis of your plot by selecting the axis, right-clicking to display the context menu, selecting **Properties** to display the property sheet, and typing `Year` in the **Title** field.

Place a title on the temperature axis of your plot by selecting the axis, displaying the property sheet, and entering the following in the **Title** field:

```
Temperature Anomaly (!Uo!NC)
```

Annotate your plot by selecting the Text Annotation tool, clicking near the top of the plot, and typing `El Nino`.

Add the special character to the annotation by selecting the annotation text, displaying the property sheet, selecting the lower-case `n` in `Nino` in the **Title** field, and replacing it with the following:

```
!Z(U+0F1)
```

Note

U+0F1 is unicode for the ñ character.

The following figure displays the output of this example:

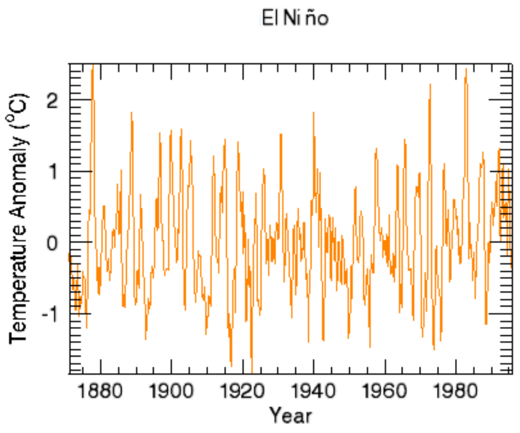


Figure 3-8: El Niño iPlot Example

Example 2

This example shows how to use the **File** → **Open** command to load binary data into the iPlot tool.

At the IDL Command Line, enter:

```
IPLOT
```

Select **File** → **Open** command to display the Open dialog, then browse to find `dirty_sine.dat` in the `examples/data` directory in the IDL distribution, and click **Open**.

In the Binary Template dialog, click **New Field**, and enter the following information in the New Field dialog:

- **Field Name:** `data` (or a name of your choosing)
- **Type:** `Byte (unsigned 8-bits)`
- **Number of Dimensions:** `1`
- **1st Dimension Size:** `256`

Click **OK** to close the New Field dialog and the Binary Template dialog, and the surface is displayed.

Note

For more information on using the Binary Template to import data, see "Using the BINARY_TEMPLATE Function" in Chapter 15 of the *Using IDL* manual.

Annotate your plot by selecting the Text Annotation tool, clicking near the curve, and typing `Noisy Sine Wave`.

The following figure displays the output of this example:

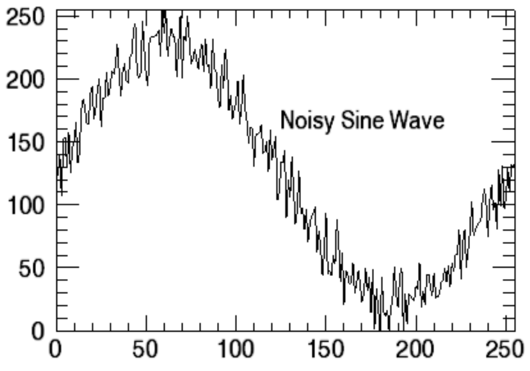


Figure 3-9: Noisy Sine Data iPlot Example

Example 3

This example shows how to use the **File** → **Import** command to load ASCII data into the iPlot tool.

At the IDL Command Line, enter:

```
IPLOT
```

Select **File** → **Import** to display the IDL Import Data wizard.

1. At Step 1, select **From a File** and click **Next>>**.
2. At Step 2, under **File Name:**, browse to find `sine_waves.txt` in the `examples/data` directory in the IDL distribution, and click **Next**>>.
3. At Step 3, select **Plot** and click **Finish**.

Then, the ASCII Template wizard is displayed.

1. At Step 1, click **Next>>** to accept the displayed data type/range definition.
2. At Step 2, click **Next**>> to accept the displayed delimiter/fields definition.
3. At Step 3, click **Finish** to accept the displayed field specification. The `sine_waves.txt` plot is displayed in the iPlot window.

The plot consists of two overlapping sine waves. To make it easier to distinguish between the two, change the appearance of the noisy sine wave to a dotted line pattern by selecting the noisy sine wave, right-clicking to display the context menu, selecting **Properties**, and changing the **Linestyle** property to a dotted line.

The following figure displays the output of this example:

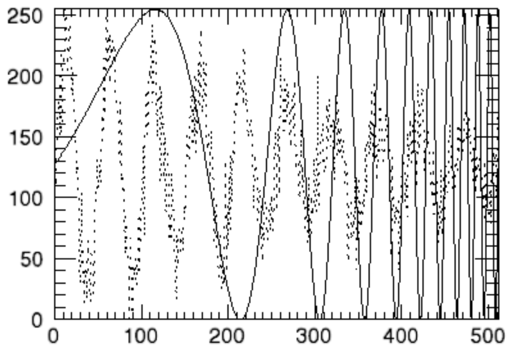


Figure 3-10: Overlapping Sine Waves iPlot Example

Version History

Introduced: 6.0

ISURFACE

The ISURFACE procedure creates an iTool and the associated user interface (UI) configured to display and manipulate surface data.

Note

If no arguments are specified, the ISURFACE procedure creates an empty Surface tool.

This routine is written in the IDL language. Its source code can be found in the file `isurface.pro` in the `lib/itools` subdirectory of the IDL distribution.

Syntax

ISURFACE[, *Z* [, *X*, *Y*]]

iTool Common Keywords: [, DIMENSIONS=[*x*, *y*]] [, IDENTIFIER=*variable*] [, LOCATION=[*x*, *y*]] [, NAME=*string*] [, OVERPLOT=*iToolID*] [, TITLE=*string*] [, VIEW_GRID=[*columns*, rows]] [, /VIEW_NEXT] [, VIEW_NUMBER=*integer*] [, {X | Y | Z}RANGE=[*min*, *max*]]

iTool Surface Keywords: [, RGB_TABLE=*array of 256 by 3 or 3 by 256 elements*] [, TEXTURE_ALPHA=*2-D array*] [, TEXTURE_BLUE=*2-D array*] [, TEXTURE_GREEN=*2-D array*] [, TEXTURE_IMAGE=*array*] [, TEXTURE_RED=*2-D array*]

Surface Object Keywords: [, BOTTOM=*index or RGB vector*] [, CLIP_PLANES=*array*] [, COLOR=*RGB vector*] [, DEPTH_OFFSET=*value*] [, /EXTENDED_LEGO] [, /HIDDEN_LINES] [, /HIDE] [, LINESTYLE=*value*] [, SHADING={0 | 1}] [, /SHOW_SKIRT] [, SKIRT=*Z value*] [, STYLE={0 | 1 | 2 | 3 | 4 | 5 | 6}] [, /TEXTURE_HIGHRES] [, /TEXTURE_INTERP] [, THICK=*points*{1.0 to 10.0}] [, /USE_TRIANGLES] [, VERT_COLORS=*vector or 2-D array*] [, ZERO_OPACITY_SKIP={0 | 1}]

Axis Object Keywords: [, {X | Y | Z}GRIDSTYLE={0 | 1 | 2 | 3 | 4 | 5 | 6}] [, {X | Y | Z}MAJOR=*integer*] [, {X | Y | Z}MINOR=*integer*] [, {X | Y | Z}SUBTICKLEN=*ratio*] [, {X | Y | Z}TEXT_COLOR=*RGB vector*] [, {X | Y | Z}TICKFONT_INDEX={0 | 1 | 2 | 3 | 4}] [, {X | Y | Z}TICKFONT_SIZE=*integer*] [, {X | Y | Z}TICKFONT_STYLE={0 | 1 | 2 | 3}] [, {X | Y | Z}TICKFORMAT=*string or string array*] [, {X | Y | Z}TICKINTERVAL=*value*] [, {X | Y | Z}TICKLAYOUT={0 | 1 | 2}] [, {X | Y | Z}TICKLEN=*value*] [, {X | Y | Z}TICKNAME=*string array*]

[, {X | Y | Z}TICKUNITS=*string*] [, {X | Y | Z}TICKVALUES=*vector*]
[, {X | Y | Z}TITLE=*string*]

Arguments

X

A vector or two-dimensional array specifying the *x*-coordinates of the grid.

If *X* is a vector:

- If *Y* and *Z* are vectors and have the same length as *X*:

  Each element of *X* specifies the *x*-coordinates of a point in space (e.g., *X*[0] specifies the *x*-coordinate for *Y*[0] and *Z*[0]). The gridding wizard will automatically launch in this case.

- If *Z* is a two-dimensional array:

  Each element of *X* specifies the *x*-coordinates for a column of *Z* (e.g., *X*[0] specifies the *x*-coordinate for *Z*[0, *]).

If *X* is a two-dimensional array, each element of *X* specifies the *x*-coordinate of the corresponding point in *Z* (X_{ij} specifies the x-coordinate of Z_{ij}).

Y

A vector or two-dimensional array specifying the *y*-coordinates of the grid.

If *Y* is a vector:

- If *X* and *Z* are vectors and have the same length as *Y*:

  Each element of *Y* specifies the *y*-coordinates of a point in space (e.g., *Y*[0] specifies the *y*-coordinate for *X*[0] and *Z*[0]). The gridding wizard will automatically launch in this case.

- If *Z* is a two-dimensional array:

  Each element of *Y* specifies the *y*-coordinates for a column of *Z* (e.g., *Y*[0] specifies the *y*-coordinate for *Z*[*, 0]).

If *Y* is a two-dimensional array, each element of *Y* specifies the *y*-coordinate of the corresponding point in *Z* (Y_{ij} specifies the y-coordinate of Z_{ij}).

Z

A vector or two-dimensional array specifying the data to be displayed.

If Z is a vector,

- If *X* and *Y* are vectors and have the same length as *Z*:

  Each element of *Z* specifies the *z*-coordinates of a point in space (e.g., *Z*[0] specifies the *z*-coordinate for *X*[0] and *Y*[0]). The gridding wizard will automatically launch in this case.

If Z is a two-dimensional array,

- If *X* and *Y* are provided:

  The surface is defined as a function of the (*x*, *y*) locations specified by their contents.

- If *X* and *Y* are not provided:

  The surface is generated as a function of the array indices of each element of *Z*.

Keywords

Note

Because keywords to the ISURFACE routine correspond to the names of registered properties of the iSurface tool, the keyword names must be specified in full, without abbreviation.

BOTTOM

Set this keyword to an RGB color for drawing the bottom of the surface. Set this keyword to a scalar to draw the bottom with the same color as the top.

CLIP_PLANES

Set this keyword to an array of dimensions [4, *N*] specifying the coefficients of the clipping planes to be applied to this object. The four coefficients for each clipping plane are of the form [A, B, C, D], where Ax + By + Cz + D = 0. Portions of this object that fall in the half space Ax + By + Cz + D > 0 will be clipped. By default, the value of this keyword is a scalar (-1) indicating that no clipping planes are to be applied.

Note

A window is only able to support a limited number of clipping planes. Some of these clipping planes may already be in use by the tool to support specific data display features. If the total number of clipping planes exceeds the limit, an informational message is displayed.

COLOR

Set this keyword to the color to be used as the foreground color for this model. The color is specified as an RGB vector. The default is [225, 184, 0].

DIMENSIONS

Set this keyword to a two-element vector of the form [width, height] to specify the dimensions of the drawing area of the specific tool in device units. The minimum width of the window correlates to the width of the menubar. The minimum window height is 100 pixels.

DEPTH_OFFSET

Set this keyword to an integer value that specifies an offset in depth to be used when rendering filled primitives. This offset is applied along the viewing axis, with positive values moving the primitive away from the viewer.

The units are “Z-Buffer units,” where a value of 1 is used to specify a distance that corresponds to a single step in the device’s Z-Buffer.

Use DEPTH_OFFSET to always cause a filled primitive to be rendered slightly deeper than other primitives, independent of model transforms. This is useful for avoiding stitching artifacts caused by rendering lines or polygons on top of other polygons at the same depth.

Note

RSI suggests using this feature to remove stitching artifacts and not as a means for “layering” complex scenes with multiple DEPTH_OFFSET values. It is safest to use only a DEPTH_OFFSET value of 0, the default, and one other non-zero value, such as 1. Many system-level graphics drivers are not consistent in their handling of DEPTH_OFFSET values, particularly when multiple non-zero values are used. This can lead to portability problems because a set of DEPTH_OFFSET values may produce better results on one machine than on another. Using IDL’s software renderer will help improve the cross-platform consistency of scenes that use DEPTH_OFFSET.

Note

DEPTH_OFFSET has no effect unless the value of the STYLE keyword is 2 or 6 (Filled or LegoFilled).

EXTENDED_LEGO

Set this keyword to force the iSurface tool to display the last row and column of data when lego display styles are selected.

HIDDEN_LINES

Set this keyword to draw point and wireframe surfaces using hidden line (point) removal. By default, hidden line removal is disabled.

HIDE

Set this keyword to a boolean value indicating whether this object should be drawn:

- 0 = Draw graphic (the default)
- 1 = Do not draw graphic

IDENTIFIER

Set this keyword to a named IDL variable that will contain the iToolID for the created tool. This value can then be used to reference this tool during overplotting operations or command-line-based tool management operations.

LINESTYLE

Set this keyword to indicate the line style that should be used to draw the surface lines. The value can be either an integer value specifying a pre-defined line style, or a two-element vector specifying a stippling pattern.

To use a pre-defined line style, set the LINESTYLE keyword equal to one of the following integer values:

- 0 = Solid line (the default)
- 1 = dotted
- 2 = dashed
- 3 = dash dot

- 4 = dash dot dot dot
- 5 = long dash
- 6 = no line drawn

To define your own stippling pattern, specify a two-element vector [*repeat*, *bitmask*], where *repeat* indicates the number of times consecutive runs of 1's or 0's in the *bitmask* should be repeated. (That is, if three consecutive 0's appear in the *bitmask* and the value of *repeat* is 2, then the line that is drawn will have six consecutive bits turned off.) The value of *repeat* must be in the range 1 ≤ *repeat* ≤ 255.

The *bitmask* indicates which pixels are drawn and which are not along the length of the line. *Bitmask* is most conveniently specified as a 16-bit hexadecimal value.

For example, `LINESTYLE = [2, 'F0F0'X]` describes a dashed line (8 bits on, 8 bits off, 8 bits on, 8 bits off).

LOCATION

Set this keyword to a two-element vector of the form [*x*, *y*] to specify the location of the upper left-hand corner of the tool relative to the display screen, in device units.

NAME

Set this keyword to a string to specify the name for this particular tool. The name is used for tool-related display purposes only–as the root of the hierarchy shown in the Tool Browser, for example.

OVERPLOT

Set this keyword to an iToolID to direct the graphical output of the particular tool to the tool specified by the provided iToolID.

Set this keyword equal to one to place the graphical output for the command in the current tool. If no current tool exists, a new tool is created.

RGB_TABLE

Set this keyword to a two-dimensional array containing RGB triplets defining the colors to be used in a color indexed texture image or by vertex colors. The values should be within the range of 0 ≤ *value* ≤ 255. The array must be a 3 by *N* array where *m* must not exceed 256.

SHADING

Set this keyword to an integer representing the type of shading to use if STYLE is set to 2 (Filled).

- 0 = Flat (default): The color has a constant intensity for each face of the surface, based on the normal vector.
- 1 = Gouraud: The colors are interpolated between vertices, and then along scanlines from each of the edge intensities.

Gouraud shading may be slower than flat shading, but results in a smoother appearance.

SHOW_SKIRT

Set this keyword to enable skirt drawing. The default is to disable skirt drawing.

SKIRT

Set this keyword to the *Z* value at which a skirt is to be defined around the array. The *Z* value is expressed in data units; the default is 0.0. If a skirt is defined, each point on the four edges of the surface is connected to a point on the skirt which has the given *Z* value, and the same *X* and *Y* values as the edge point. In addition, each point on the skirt is connected to its neighbor. The skirt value is ignored if skirt drawing is disabled (see SHOW_SKIRT above). IDL converts, maintains, and returns this data as double-precision floating-point.

STYLE

Set this keyword to and integer value that indicates the style to be used to draw the surface. Valid values are:

- 0 = Points
- 1 = Wire mesh
- 2 = Filled (the default)
- 3 = RuledXZ
- 4 = RuledYZ
- 5 = Lego
- 6 = LegoFilled: for outline or shaded and stacked histogram-style plots.

TEXTURE_ALPHA

Set the keyword to a two-dimensional array containing the alpha channel of an image to be used as a texture image. Use of this keyword requires that TEXTURE_RED, TEXTURE_GREEN, and TEXTURE_BLUE be set to arrays of identical size and type.

TEXTURE_BLUE

Set the keyword to a two-dimensional array containing the blue channel of an image to be used as a texture image. Use of this keyword requires that TEXTURE_RED and TEXTURE_GREEN be set to arrays of identical size and type.

TEXTURE_GREEN

Set the keyword to a two-dimensional array containing the green channel of an image to be used as a texture image. Use of this keyword requires that TEXTURE_RED and TEXTURE_BLUE be set to arrays of identical size and type.

TEXTURE_HIGHRES

Set this keyword to cause texture tiling to be used as necessary to maintain the full pixel resolution of the original texture image.

Setting this keyword is recommended if IDL is running on modern 3-D hardware and resolution loss due to downscaling becomes problematic. If not set, and the texture map is larger than the maximum resolution supported by the 3-D hardware, the texture is scaled down to the maximum resolution supported by the 3-D hardware on your system. The default value is 0.

TEXTURE_IMAGE

Set this keyword to an array containing an image to be texture mapped onto the surface. If this keyword is omitted or set to a null object reference, no texture map is applied and the surface is filled with the color specified by the COLOR or VERTEX_COLORS property. The image array can be a two-dimensional array of color indexes or a three-dimensional array specifying RGB values at each pixel (3 x *n* x *m*, *n* x 3 x *m*, or *n* x *m* x 3). Setting TEXTURE_IMAGE to a three-dimensional array contains an alpha channel (4 x *n* x *m*, *n* x 4 x *m*, or *n* x *m* x 4) allows you to create a transparent iSurface object. The TEXTURE_IMAGE keyword will override any values passed to TEXTURE_RED, TEXTURE_GREEN, TEXTURE_BLUE, or TEXTURE_ALPHA.

TEXTURE_INTERP

Set this keyword to a nonzero value to indicate that bilinear sampling is to be used with texture mapping. The default method is nearest-neighbor sampling.

TEXTURE_RED

Set the keyword to a two-dimensional array containing the red channel of an image to be used as a texture image. Use of this keyword requires that TEXTURE_GREEN and TEXTURE_BLUE be set to arrays of identical size and type.

THICK

Set this keyword to a value between 1.0 and 10.0, specifying the line thickness to use to draw surface lines, in points. The default is 1.0 points.

TITLE

Set this keyword to a string to specify a title for the tool. The title is displayed in the title bar of the tool.

USE_TRIANGLES

Set this keyword to force the iSurface tool to use triangles instead of quads to draw the surface and skirt.

VERT_COLORS

Set this keyword to a vector, two-dimensional array of equal size to *Z*, or a two-dimensional array containing RGB triplets representing colors to be used at each vertex. If this keyword is set to a vector or a two-dimensional array of equal size to *Z*, these values are indices into a color table that can be specified by the RGB_TABLE keyword. If the RGB_TABLE keyword is not set, a grayscale color is used. If more vertices exist than elements in VERT_COLORS, the elements of VERT_COLORS are cyclically repeated. If this keyword is omitted, the surface is drawn in the color specified by the COLOR keyword or the default color.

VIEW_GRID

Set this keyword to a two-element vector of the form [*columns*, *rows*] to specify the view layout within the new tool. This keyword is only used if a new tool is being created (for example, if OVERPLOT, VIEW_NEXT, or VIEW_NUMBER are specified then VIEW_GRID is ignored).

VIEW_NEXT

Set this keyword to change the view selection to the next view following the currently-selected view before issuing any graphical commands. If the currently-selected view is the last one in the layout, then /VIEW_NEXT will cause the first view in the layout to become selected. This keyword is ignored if no current tool exists.

Note

The contents of the newly-selected view will be emptied unless /OVERPLOT is set.

VIEW_NUMBER

Set this keyword to change the currently-selected view to the view specified by the VIEW_NUMBER before issuing any graphical commands. The view number starts at 1, and corresponds to the position of the view within the graphics container (not necessarily the position on the screen). This keyword is ignored if no current tool exists.

Note

The contents of the newly-selected view will be emptied unless /OVERPLOT is set.

[XYZ]MAJOR

Set this keyword to an integer representing the number of major tick marks. The default is -1, specifying that IDL will compute the number of tickmarks. Setting MAJOR equal to zero suppresses major tickmarks entirely.

[XYZ]MINOR

Set this keyword to an integer representing the number of minor tick marks. The default is -1, specifying that IDL will compute the number of tickmarks. Setting MINOR equal to zero suppresses minor tickmarks entirely.

[XYZ]RANGE

Set this keyword to the desired data range of the axis, a 2-element vector. The first element is the axis minimum, and the second is the maximum.

[XYZ]SUBTICKLEN

Set this keyword to a floating-point scale ratio specifying the length of minor tick marks relative to the length of major tick marks. The default is 0.5, specifying that the minor tick mark is one-half the length of the major tick mark.

[XYZ]TEXT_COLOR

Set this keyword to an RGB value specifying the color for the axis text. The default value is [0, 0, 0] (black).

[XYZ]TICKFONT_INDEX

Set this keyword equal to one of the following integers, which represent the type of font to be used for the axis text:

- 0 = Helvetica
- 1 = Courier
- 2 = Times
- 3 = Symbol
- 4 = Hershey

[XYZ]TICKFONT_SIZE

Set this keyword to an integer representing the point size of the font used for the axis text. The default is 12.0 points.

[XYZ]TICKFONT_STYLE

Set this keyword equal to one of the following integers, which represent the style of font to be used for the axis text:

- 0 = Normal
- 1 = Bold
- 2 = Italic
- 3 = Bold Italic

[XYZ]TICKFORMAT

Set this keyword to a string, or an array of strings, in which each string represents a format string or the name of a function to be used to format the tick mark labels. If an array is provided, each string corresponds to a level of the axis. The TICKUNITS keyword determines the number of levels for an axis.

If the string begins with an open parenthesis, it is treated as a standard format string. See “Format Codes” in Chapter 10 of the *Building IDL Applications* manual.

If the string does not begin with an open parenthesis, it is interpreted as the name of a callback function to be used to generate tick mark labels.

If TICKUNITS are not specified:

- The callback function is called with three parameters: *Axis*, *Index*, and *Value*, where:
- *Axis* is the axis number: 0 for X axis, 1 for Y axis, 2 for Z axis
- *Index* is the tick mark index (indices start at 0)
- *Value* is the data value at the tick mark (a double-precision floating point value)

If TICKUNITS are specified:

The callback function is called with four parameters: *Axis*, *Index*, *Value*, and *Level*, where:

- *Axis*, *Index*, and *Value* are the same as described above.
- *Level* is the index of the axis level for the current tick value to be labeled. (Level indices start at 0.)

Used with the LABEL_DATE function, this property can easily create axes with date/time labels.

[XYZ]TICKINTERVAL

Set this keyword to a floating-point scalar indicating the interval between major tick marks for the first axis level. The default value is computed according to the axis [XYZ]RANGE and the number of major tick marks ([XYZ]MAJOR). The value of this keyword takes precedence over the value set for the [XYZ]MAJOR keyword.

For example, if TICKUNITS = ['S', 'H', 'D'], and TICKINTERVAL = 30, then the interval between major ticks for the first axis level will be 30 seconds.

[XYZ]TICKLAYOUT

Set this keyword to integer scalar that indicates the tick layout style to be used to draw each level of the axis.

Valid values include:

- 0 = The axis line, major tick marks and tick labels are all included. Minor tick marks only appear on the first level of the axis. This is the default tick layout style.
- 1 = Only the labels for the major tick marks are drawn. The axis line, major tick marks, and minor tick marks are omitted.
- 2 = Each major tick interval is outlined by a box. The tick labels are positioned within that box (left-aligned). For the first axis level only, the major and minor tick marks will also be drawn.

Note

For all tick layout styles, at least one tick label will appear on each level of the axis (even if no major tick marks fall along the axis line). If there are no major tick marks, the single tick label will be centered along the axis.

[XYZ]TICKLEN

Set this keyword to a floating-point value that specifies the length of each major tick mark, measured in data units. The recommended, and default, tick mark length is 0.2. IDL converts, maintains, and returns this data as double-precision floating-point.

[XYZ]TICKNAME

Set this keyword to a string array of up to 30 elements that controls the annotation of each tick mark.

[XYZ]TICKUNITS

Set this keyword to a string (or a vector of strings) indicating the units to be used for axis tick labeling. If more than one unit is provided, the axis will be drawn in multiple levels, one level per unit.

The order in which the strings appear in the vector determines the order in which the corresponding unit levels will be drawn. The first string corresponds to the first level (the level nearest to the primary axis line).

Valid unit strings include:

- "Numeric"
- "Years"
- "Months"
- "Days"
- "Hours"
- "Minutes"
- "Seconds"
- "Time" - Use this value to indicate that the tick values are time values; IDL will determine the appropriate time intervals and tick label formats based upon the range of values covered by the axis.
- ""- Use the empty string to indicate that no tick units are being explicitly set. This implies that a single axis level will be drawn using the "Numeric" unit. This is the default setting.

If any of the time units are utilized, then the tick values are interpreted as Julian date/time values. Note that the singular form of each of the time value strings is also acceptable (e.g, TICKUNITS = 'Day' is equivalent to TICKUNITS = 'Days').

Note

Julian values must be in the range -1095 to 1827933925, which corresponds to calendar dates 1 Jan 4716 B.C.E. and 31 Dec 5000000hidd, respectively.

[XYZ]TICKVALUES

Set this keyword to a floating-point vector of data values representing the values at each tick mark. If TICKVALUES is set to 0, the default, IDL computes the tick values based on the axis range and the number of major ticks. IDL converts, maintains, and returns this data as double-precision floating-point.

[XYZ]TITLE

Set this keyword to a string representing the title of the specified axis.

ZERO_OPACITY_SKIP

Set this keyword to gain finer control over the rendering of textured surface pixels (texels) with an opacity of 0 in the texture map. Texels with zero opacity do not affect the color of a screen pixel since they have no opacity. If this keyword is set to 1, any texels are “skipped” and not rendered at all. If this keyword is set to zero, the Z-buffer is updated for these pixels and the display image is not affected as noted above. By updating the Z-buffer without updating the display image, the surface can be used as a *clipping* surface for other graphics primitives drawn after the current graphics object. The default value for this keyword is 1.

Note

This keyword has no effect if no texture map is used or if the texture map in use does not contain an opacity channel.

Examples

In the IDL Intelligent Tools system, data can be imported from the IDL Command Line (as described in Example 1), or data can be imported via the File menu in the iTool window (as described in Examples 2 and 3). For detailed information on importing data via the iTool file menu, refer to “Data Import Methods” in Chapter 2 of the *iTool User’s Guide* manual.

Example 1

This example shows how to use the IDL Command Line to load data into the iSurface tool.

At the IDL Command Line, enter:

```
file = FILEPATH('surface.dat', $
   SUBDIRECTORY = ['examples', 'data'])
data = READ_BINARY(file, DATA_DIMS = [350, 450], DATA_TYPE = 2, $
   ENDIAN = 'little')
ISURFACE, data, TITLE = 'Maroon Bells Elevation', $
   COLOR = [255, 128, 0]
```

Place a title on the elevation axis of your plot by selecting the axis, right-clicking to display the context menu, selecting **Properties** to display the property sheet, and typing `Elevation (m)` in the **Title** field.

Use the **Operations** → **Statistics...** option to display the iTools Statistics dialog. Within this dialog, observe the `Z` value's `Maximum`, which is `4241` at `[29, 253]`. Close the iTools Statistics dialog by selecting **File** → **Close**.

Annotate your plot by selecting the Text Annotation tool, clicking near the top of the highest peak in the display, and typing `Highest Point (4241 m)`. Draw a line annotation between the text annotation and the highest peak on the surface.

The following figure displays the output of this example:

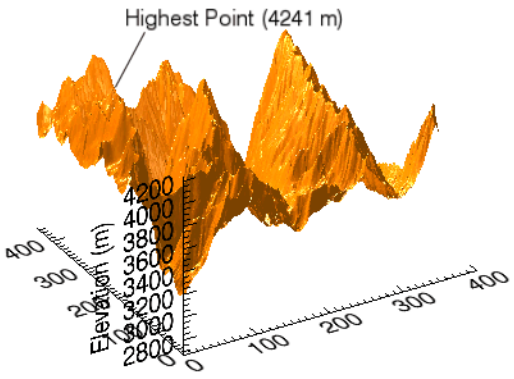


Figure 3-11: Maroon Bells iSurface Example

Example 2

This example shows how to use the **File** → **Open** command to load binary data into the iSurface tool.

At the IDL Command Line, enter:

```
ISURFACE
```

Select **File** → **Open** to display the Open dialog, then browse to find `idemosurf.dat` in the `examples/data` directory in the IDL distribution, and click **Open**.

The Binary Template wizard is displayed. In the Binary Template, change **File's byte ordering** to `Little Endian`. Then, click **New Field**, and enter the following information in the New Field dialog:

- **Field Name:** `data` (or a name of your choosing)
- **Type:** `Float (32 bit)`
- **Number of Dimensions:** `2`
- **1st Dimension Size:** `200`
- **2nd Dimension Size:** `200`

Click **OK** to close the New Field dialog and the Binary Template dialog, and the surface is displayed.

Note

For more information on using the Binary Template to import data, see "Using the BINARY_TEMPLATE Function" in Chapter 15 of the *Using IDL* manual.

Insert a contour onto the surface by clicking the **Surface Contour** button on the toolbar, then clicking and dragging on the surface to position the contour at the desired height.

The following figure displays the output of this example:

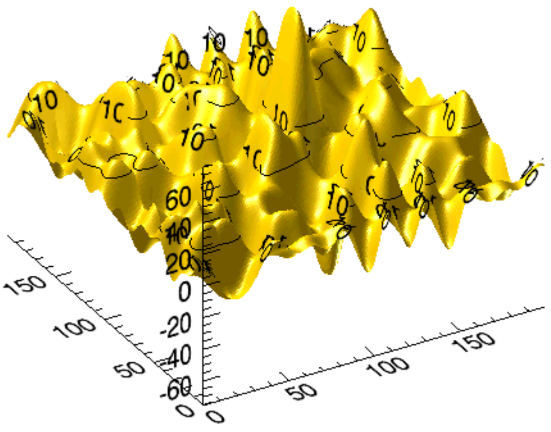


Figure 3-12: Binary Surface Data iSurface Example

Example 3

This example shows how to use the **File** → **Import** command to load ASCII data into the iSurface tool.

At the IDL Command Line, enter:

```
ISURFACE
```

Select **File** → **Import** to display the IDL Import Data wizard.

1. At Step 1, select **From a File** and click **Next>>**.
2. At Step 2, under **File Name:**, browse to find `irreg_grid1.txt` in the `examples/data` directory in the IDL distribution, and click **Next**>>.
3. At Step 3, select **Surface** and click **Finish**.

Then, the ASCII Template wizard is displayed.

1. At Step 1, click **Next>>** to accept the displayed Data Type/Range definitions.
2. At Step 2, click **Next>>** to accept the displayed Delimiter/Fields definitions.
3. At Step 3, click **Finish** to accept the displayed Field Specifications.

Note

For more information on using the ASCII Template to import data, see "Using the ASCII_TEMPLATE Function" in Chapter 14 of the *Using IDL* manual.

At the iTool's Create Visualization window, you have the option of launching the Gridding wizard or not creating a visualization. Choose **Launch the gridding wizard** and click **Ok**.

4. At Step 1, click **Next>>** to accept the interpolation of data values and locations.
5. At Step 2, click **Next>>** to accept the dimensions, start and spacing.
6. At Step 3, select **Inverse Distance** as the gridding method, click **Preview** to preview the possible results, and click **Finish** to display the surface.

Double-click the surface to display the Properties sheet, and change the **Fill shading** setting from `Flat` to `Gouraud`.

Use the Rotate tool on the Toolbar to rotate the surface slightly forward to better display the surface convolutions.

The following figure displays the output of this example.

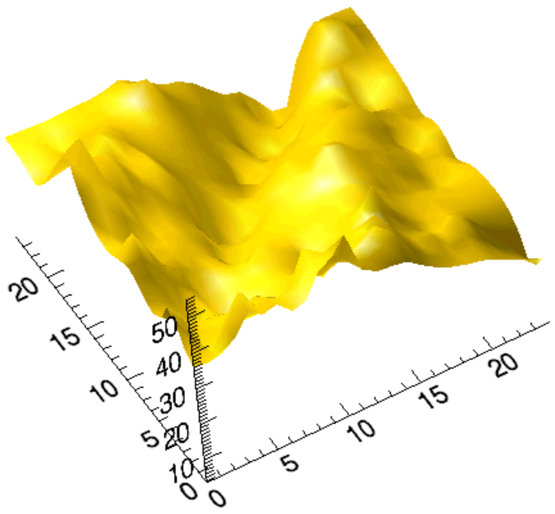


Figure 3-13: ASCII Surface Data iSurface Example

Version History

Introduced: 6.0

ITCURRENT

The ITCURRENT procedure is used to set the current tool in the IDL Intelligent Tools system. This routine is used with the identifier of the tool to make it current in the system. If the identifier is valid, the specified tool becomes current.

When a tool is set as current, the visible display or the focus state of the tool does not change. Only the internal setting of the current tool changes.

Besides using this procedure to set the current tool, a tool is made current when it is created or when it is placed in focus in the current windowing system.

This routine is written in the IDL language. Its source code can be found in the file `itcurrent.pro` in the `lib/itools` subdirectory of the IDL distribution.

Syntax

ITCURRENT, *iToolID*

Arguments

iToolID

The identifier of the existing iTool to be set as current.

Keywords

None.

Example

Enter the following at the IDL Command Line:

```
IPLOT, IDENTIFIER = PlotID1
current1 = ITGETCURRENT()
PRINT, 'The current tool is ', current1
```

An iPlot tool is created, and the newly created iPlot tool becomes the current tool. Output similar to the following appears in the IDL Output Log:

```
The current tool is /TOOLS/IPLOT_8
```

Enter the following at the IDL Command Line:

```
IPLOT, IDENTIFIER = PlotID2
current2 = ITGETCURRENT()
PRINT, 'The current tool is ', current2
```

A second iPlot tool is created, and this newly created iPlot tool becomes the current tool. Output similar to the following appears in the IDL Output Log:

```
The current tool is /TOOLS/IPLOT_9
```

Enter the following at the IDL Command Line:

```
ISURFACE, IDENTIFIER = SurfaceID1
current3 = ITGETCURRENT()
PRINT, 'The current tool is ', current3
```

An iSurface tool is created, and the newly created iSurface tool becomes the current tool. Output similar to the following appears in the IDL Output Log:

```
The current tool is /TOOLS/ISURFACE_5
```

Enter the following at the IDL Command Line:

```
ITCURRENT, PlotID1
current = ITGETCURRENT()
PRINT, 'The current tool is ', current
END
```

The iPlot tool created at the beginning of the example (PlotID1) becomes the current tool. Output similar to the following appears in the IDL Output Log:

```
The current tool is /TOOLS/IPLOT_8
```

Note that the system ID of the current tool (IPLOT_8) is the same as that of the current tool at the beginning of the exercise.

Version History

Introduced: 6.0

See Also

ITDELETE, ITGETCURRENT, ITRESET

ITDELETE

The ITDELETE procedure is used to delete a tool in the IDL Intelligent Tools system. If a valid identifier is provided, the tool represented by the identifier is destroyed. If no identifier is provided, the current tool is destroyed.

When a tool is destroyed, all resources specific to that tool are released and the tool ceases to exist.

This routine is written in the IDL language. Its source code can be found in the file `itdelete.pro` in the `lib/itools` subdirectory of the IDL distribution.

Syntax

ITDELETE[, *iToolID*]

Arguments

iToolID

This optional argument contains the identifier for the specific iTool to delete. If not provided, the current tool is destroyed.

Keywords

None.

Example

Enter the following at the IDL Command Line:

```
IPLOT, IDENTIFIER = PlotID1
ISURFACE, IDENTIFIER = SurfaceID1
```

Two tools are created: an iPlot tool and an iSurface tool.

Next, enter the following at the IDL Command Line:

```
ITDELETE, plotID1
```

The iPlot tool is deleted, leaving only the iSurface tool.

Version History

Introduced: 6.0

See Also

ITCURRENT, ITGETCURRENT, ITRESET

ITGETCURRENT

The ITGETCURRENT function is used to get the identifier of the current tool in the IDL Intelligent Tools system.

This routine is written in the IDL language. Its source code can be found in the file `itgetcurrent.pro` in the `lib/itools` subdirectory of the IDL distribution.

Syntax

Result = ITGETCURRENT()

Return Value

Returns the identifier of the current tool in the iTool system. If no tool exists, an empty string (`''`) is returned.

Arguments

None.

Keywords

None.

Example

The following example line of code creates a plot tool:

```
IPLOT, SIN(FINDGEN(361)*!DTOR), COLOR = [0, 0, 255], THICK = 2
```

The resulting plot tool contains a blue sine function, with a line thickness of 2. To overplot a cosine function on this display, the following lines of code are used:

```
idSin = ITGETCURRENT()
IPLOT, COS(FINDGEN(361)*!DTOR), COLOR = [0, 255, 0], THICK = 2, $
   OVERPLOT = idSin
```

However, it is not necessary to use ITGETCURRENT to retrieve the current tool for overplotting. The following method is also possible because the creation of a new tool causes it to be set as current in the system. In this scenario, the commands to generate the same display are:

```
IPLOT, SIN(FINDGEN(361)*!DTOR), COLOR = [0, 0, 255], THICK = 2
IPLOT, COS(FINDGEN(361)*!DTOR), COLOR = [0, 255, 0], THICK = 2, $
   /OVERPLOT
```

Version History

Introduced: 6.0

See Also

ITCURRENT, ITDELETE, ITRESET

ITREGISTER

The ITREGISTER procedure is used to register tool object classes or other iTool functionality with the IDL Intelligent Tools system.

This routine is written in the IDL language. Its source code can be found in the file `itregister.pro` in the `lib/itools` subdirectory of the IDL distribution.

Syntax

ITREGISTER, *Name*, *ItemName* [, TYPES=*string*] [, /UI_PANEL]
[, /UI_SERVICE] [, /VISUALIZATION]

Arguments

Name

A string containing the name used to refer to the associated class once registration is completed. Subsequent calls to create items of this type will use this name to identify the associated class.

ItemName

A string containing the class name of the object class or user interface routine that is to be associated with *Name*. When an item of name *Name* is requested from the system, an object of this class is created or the specified routine is called.

Keywords

Note

Keywords supplied in the call to ITREGISTER but not listed here are passed directly to the underlying objects' registration routines.

TYPES

This keyword is only used in conjunction with the UI_PANEL keyword.

Set this keyword equal to a string or string array containing iTool types with which the UI panel should be associated. When the registered type of a UI panel matches the registered type of an iTool, the panel will be displayed as part of the iTool's interface.

UI_PANEL

Set this keyword to indicate that a UI panel is being registered with the system. When this keyword is set, the value of *Name* is the string used to refer to the panel and *ItemName* is the routine that should be called when the panel is created.

To specify that the UI panel is associated with a particular iTool or iTools, set the TYPES keyword to the iTool types that should expose this panel.

UI_SERVICE

Set this keyword to indicate that a UI service is being registered with the system. When this keyword is set, the value of *Name* is the string used to refer to the UI service and *ItemName* is the routine that should be called to execute the service.

VISUALIZATION

Set this keyword to indicate that a visualization is being registered with the system. When this keyword is set, the value of *Name* is the string used to refer to the visualization type, and *ItemName* is the name of the visualization type's class definition routine.

Examples

Suppose you have an iTool class definition file named `myTool__define.pro`, located in a directory included in IDL's !PATH system variable. Register this class with the iTool system with the following command:

```
ITREGISTER, 'My First Tool', 'myTool'
```

Tools defined by the `myTool` class definition file can now be created by the iTool system by specifying the tool name `My First Tool`.

Similarly, suppose you have a user interface service defined in a file named `myUIFileOpen.pro`. Register this UI service with the iTool system with the following command:

```
ITREGISTER, 'My File Open', 'myUIFileOpen', /UI_SERVICE
```

Finally, suppose you have a user interface panel defined in a file named `myPanel.pro`, and that you want this panel to be added to the user interface of iTools registered with the TYPES property set to `MYTOOL`. Register this UI panel with the iTool system with the following command:

```
ITREGISTER, 'My Panel', 'myPanel', /UI_PANEL, TYPES = 'MYTOOL'
```

Version History

Introduced: 6.0

See Also

Chapter 5, “Creating an iTool” in the *iTool Developer’s Guide* manual.

ITRESET

The ITRESET procedure resets the IDL iTools session. When called, all active tools and overall system management is destroyed and associated resources released.

This class is written in the IDL language. Its source code can be found in the file `itreset.pro` in the `lib/itools` subdirectory of the IDL distribution.

Syntax

ITRESET[, /NO_PROMPT]

Arguments

None

Keywords

NO_PROMPT

Set this keyword to disable prompting the user before resetting the system. If this keyword is set, the user is not presented with a prompt and the reset is performed immediately.

Examples

The iTool Data Manager system maintains your data during the entire IDL session, unless ITRESET is used. This example shows how the data is maintained and how ITRESET is used to clear the iTool Data Manager.

Read in plot data and load it into an iPlot tool at the IDL Command Line:

```
file = FILEPATH('dirty_sine.dat', $
   SUBDIRECTORY = ['examples', 'data'])
data = READ_BINARY(file, DATA_DIMS = [256, 1])
IPLOT, data
```

Delete this tool with the ITDELETE procedure at the IDL Command Line:

```
ITDELETE
```

Read in surface data and load it into an iSurface tool at the IDL Command Line:

```
file = FILEPATH('elevbin.dat', $
   SUBDIRECTORY = ['examples', 'data'])
data = READ_BINARY(file, DATA_DIMS = [64, 64])
ISURFACE, data
```

Use **Window** → **Data Manager...** to access the Data Manager Browser. The browser contains both plot and surface parameters. Although the iPlot tool was deleted, its data remains in the Data Manager. Click **Dismiss**.

Use **File** → **New** → **iPlot** to create an empty iPlot tool. If you want to load the plot data in the Data Manager into this tool, use **Insert** → **Visualization** to access the Insert Visualization dialog, which allows you to specify the plot data to be displayed.

At the IDL Command Line, enter:

```
ITRESET, /NO_PROMPT
```

The two iTools are deleted and the data in the Data Manager is released. To verify the data in released, create an empty iSurface tool at the IDL Command Line:

```
ISURFACE
```

Use **Window** → **Data Manager...** to access the Data Manager Browser. No data appears in the browser. The iTool Data Manger in empty. Click **Dismiss**.

At the IDL Command Line, enter:

```
ITRESET, /NO_PROMPT
```

Version History

Introduced: 6.0

See Also

ITCURRENT, ITDELETE, ITGETCURRENT

IVOLUME

The IVOLUME procedure creates an iTool and associated user interface (UI) configured to display and manipulate volume data.

Note

If no arguments are specified, the IVOLUME procedure creates an empty Volume tool.

This routine is written in the IDL language. Its source code can be found in the file `ivolume.pro` in the `lib/itools` subdirectory of the IDL distribution.

Syntax

IVOLUME[, *Vol*$_0$[, *Vol*$_1$][, *Vol*$_2$, *Vol*$_3$]]

iTool Common Keywords: [, DIMENSIONS=[*x*, *y*]] [, IDENTIFIER=*variable*] [, LOCATION=[*x*, *y*]] [, NAME=*string*] [, OVERPLOT=*iToolID*] [, TITLE=*string*] [, VIEW_GRID=[*columns*, *rows*]] [, /VIEW_NEXT] [. VIEW_NUMBER=*integer*] [, {X | Y | Z}RANGE=[*min*, *max*]]

iTool Volume Keywords: [, /AUTO_RENDER]
[, RENDER_EXTENTS={0 | 1 | 2}] [, RENDER_QUALITY={1 | 2}]
[, SUBVOLUME=[*xmin*, *ymin*, *zmin*, *xmax*, *ymax*, *zmax*]]
[, VOLUME_DIMENSIONS=[*width*, *height*, *depth*]]
[, VOLUME_LOCATION=[*x*, *y*, *z*]]

Volume Object Keywords: [, AMBIENT=*RGB vector*]
[, BOUNDS=[*xmin*, *ymin*, *zmin*, *xmax*, *ymax*, *zmax*]] [, CLIP_PLANES=*array*]
[, COMPOSITE_FUNCTION={0 | 1 | 2 | 3}] [, CUTTING_PLANES=*array*]
[, DEPTH_CUE=[*zbright*, *zdim*]] [, /HIDE] [, HINTS={0 | 1 | 2 | 3}]
[, /INTERPOLATE] [, /LIGHTING_MODEL] [, OPACITY_TABLE0=*byte array of 256 elements*] [, OPACITY_TABLE1=*byte array of 256 elements*]
[, RENDER_STEP=[*x*, *y*, *z*]] [, RGB_TABLE0=*byte array of 256 by 3 or 3 by 256 elements*] [, RGB_TABLE1=*byte array of 256 by 3 or 3 by 256 elements*]
[, /TWO_SIDED] [, /ZBUFFER] [, ZERO_OPACITY_SKIP={0 | 1}]

Axis Object Keywords: [, {X | Y | Z}GRIDSTYLE={0 | 1 | 2 | 3 | 4 | 5 | 6}] [, {X | Y | Z}MAJOR=*integer*] [, {X | Y | Z}MINOR=*integer*] [, {X | Y | Z}SUBTICKLEN=*ratio*] [, {X | Y | Z}TEXT_COLOR=*RGB vector*] [, {X | Y | Z}TICKFONT_INDEX={0 | 1 | 2 | 3 | 4}] [, {X | Y | Z}TICKFONT_SIZE=*integer*] [, {X | Y | Z}TICKFONT_STYLE={0 | 1 | 2 | 3}] [, {X | Y | Z}TICKFORMAT=*string or string array*] [, {X | Y | Z}TICKINTERVAL=*value*] [, {X | Y | Z}TICKLAYOUT={0 | 1 | 2}] [, {X | Y | Z}TICKLEN=*value*] [, {X | Y | Z}TICKNAME=*string array*] [, {X | Y | Z}TICKUNITS=*string*] [, {X | Y | Z}TICKVALUES=*vector*] [, {X | Y | Z}TITLE=*string*]

Arguments

Note

The volume data provided in the Vol_0, Vol_1, Vol_2, and Vol_3 arguments are scaled into byte values (ranging from 0 to 255) with the BYTSCL function to facilitate using the volume data as indices into the RGB and OPACITY tables. This scaling is done for display purposes only; the iVolume tool maintains the original data as supplied with the arguments for use in other operations. The minimum and maximum values used by the BYTSCL function may be adjusted in the volume’s property sheet. By default, the tool uses the minimum and maximum values of all volume parameters to uniformly byte-scale the data.

Vol_0, Vol_1, Vol_2, Vol_3

A three-dimensional array of any numeric type containing volume data. Arrays of strings, structures, object references, and pointers are not allowed. If more than one volume is specified, they must all have the same dimensions.

The number of volumes present and the value of the COMPOSITE_FUNCTION keyword determine how the volume data is rendered by the iVolume tool. The number of volume arguments determine how the `src` and `srcalpha` values for the COMPOSITE_FUNCTION are computed:

- If Vol_0 is the only argument present, the values of `src` and `srcalpha` are taken directly from the RGB and OPACITY tables, as indexed by each volume data sample:

```
src = RGB_TABLE0[VOL0]
srcalpha = OPACITY_TABLE0[VOL0]
```

- If *Vol*$_0$ and *Vol*$_1$ are the only arguments present, the two volumes are blended together using independent tables:

```
src = (RGB_TABLE0[VOL0]*RGB_TABLE1[VOL1])/256
srcalpha = (OPACITY_TABLE0[VOL0]*OPACITY_TABLE1[VOL1])/256
```

- If all the arguments are present, *Vol*$_0$ indexes the red channel of RGB_TABLE0, *Vol*$_1$ indexes the green channel of RGB_TABLE0, and *Vol*$_2$ indexes the blue channel of RGB_TABLE0. The *Vol*$_3$ argument indexes OPACITY_TABLE0:

```
src = (RGB_TABLE[VOL0, 0], RGB_TABLE[VOL1, 1], $
   RGB_TABLE[VOL2, 2])/256
srcalpha = (OPACITY_TABLE0[VOL3])/256.
```

Note

If all the arguments are present, the composite function cannot be set to the average-intensity projection (COMPOSITE_FUNCTION = 3).

Keywords

Note

Because keywords to the IVOLUME routine correspond to the names of registered properties of the iVolume tool, the keyword names must be specified in full, without abbreviation.

AMBIENT

Use this keyword to set the color and intensity of the volume's base ambient lighting. Color is specified as an RGB vector. The default is [255, 255, 255]. AMBIENT is applicable only when LIGHTING_MODEL is set.

AUTO_RENDER

Set this keyword to 1 to always render the volume. The default is to not render the volume each time the tool window is drawn.

BOUNDS

Set this keyword to a six-element vector of the form [x_{min}, y_{min}, z_{min}, x_{max}, y_{max}, z_{max}], which represents the sub-volume to be rendered. This keyword is the same as the SUBVOLUME keyword.

CLIP_PLANES

Set this keyword to an array of dimensions [4, *N*] specifying the coefficients of the clipping planes to be applied to this object. The four coefficients for each clipping plane are of the form [A, B, C, D], where Ax + By + Cz + D = 0. Portions of this object that fall in the half space Ax + By + Cz + D > 0 will be clipped. By default, the value of this keyword is a scalar (-1) indicating that no clipping planes are to be applied.

Note

Clipping planes are equivalent to cutting planes (refer to the CUTTING_PLANES keyword). The CUTTING_PLANES will be applied first, then the CLIP_PLANES (until a maximum number of planes is reached).

Note

A window is only able to support a limited number of clipping planes. Some of these clipping planes may already be in use by the tool to support specific data display features. If the total number of clipping planes exceeds the limit, an informational message is displayed.

COMPOSITE_FUNCTION

The composite function determines the value of a pixel on the viewing plane by analyzing the voxels falling along the corresponding ray, according to one of the following compositing functions:

- 0 = Alpha (default): Alpha-blending. The recursive equation

  ```
  dest' = src * srcalpha + dest * (1 - srcalpha)
  ```

  is used to compute the final pixel color.

- 1 = MIP: Maximum intensity projection. The value of each pixel on the viewing plane is set to the brightest voxel, as determined by its opacity. The most opaque voxel’s color appropriation is then reflected by the pixel on the viewing plane.

- 2 = Alpha sum: Alpha-blending. The recursive equation

  ```
  dest' = src + dest * (1 - srcalpha)
  ```

  is used to compute the final pixel color. This equation assumes that the color tables have been pre-multiplied by the opacity tables. The accumulated values can be no greater than 255.
- 3 = Average: Average-intensity projection. The resulting image is the average of all voxels along the corresponding ray.

Note

This option (COMPOSITE_FUNCTION = 3) is not supported for 4-channel volumes.

CUTTING_PLANES

Set this keyword to a floating-point array with dimensions (4, *n*) specifying the coefficients of *n* cutting planes. The cutting plane coefficients are in the form $\{\{n_x, n_y, n_z, D\}, ...\}$ where $(n_x)X+(n_y)Y+(n_z)Z+ D > 0$, and (X, Y, Z) are the voxel coordinates. To clear the cutting planes, set this property to any scalar value (e.g. CUTTING_PLANES = 0). By default, no cutting planes are defined.

DEPTH_CUE

Set this keyword to a two-element floating-point array [*zbright*, *zdim*] specifying the near and far Z planes between which depth cueing is in effect.

Depth cueing causes an object to appear to fade into the background color of the view object with changes in depth. If the depth of an object is further than *zdim* (that is, if the object's location in the Z direction is farther from the origin than the value specified by *zdim*), the object will be painted in the background color.

Similarly, if the object is closer than the value of *zbright*, the object will appear in its "normal" color. Anywhere in-between, the object will be a blend of the background color and the object color. For example, if the DEPTH_CUE property is set to [-1, 1], an object at the depth of 0.0 will appear as a 50% blend of the object color and the view color.

The relationship between Z_{bright} and Z_{dim} determines the result of the rendering:

- $Z_{bright} < Z_{dim}$: Rendering darkens with depth.
- $Z_{bright} > Z_{dim}$: Rendering brightens with depth.
- $Z_{bright} = Z_{dim}$: Disables depth cueing.

You can disable depth cueing by setting $z_{bright} = z_{dim}$. The default is [0.0, 0.0].

DIMENSIONS

Set this keyword to a two-element vector of the form [width, height] to specify the dimensions of the drawing area of the specific tool in device units. The minimum width of the window correlates to the width of the menubar. The minimum window height is 100 pixels.

HIDE

Set this keyword to a boolean value indicating whether the volume should be drawn:

- 0 = Draw graphic (the default)
- 1 = Do not draw graphic

HINTS

Set this keyword to specify one of the following acceleration hints:

- 0 = Disables all acceleration hints (default).
- 1 = Enables Euclidean distance map (EDM) acceleration. This option generates a volume map containing the distance from any voxel to the nearest non-zero opacity voxel. The map is used to speed ray casting by allowing the ray to jump over open spaces. It is most useful with sparse volumes. After setting the EDM hint, the draw operation generates the volume map; this process can take some time. Subsequent draw operations will reuse the generated map and may be much faster, depending on the volume’s sparseness. A new map is not automatically generated to match changes in opacity tables or volume data (for performance reasons). The user may force recomputation of the EDM map by setting the HINTS property to 1 again.
- 2 = Enables the use of multiple CPUs for volume rendering if the platforms used support such use. If HINTS is set to 2, IDL will use all the available (up to 8) CPUs to render portions of the volume in parallel.
- 3 = Selects the two acceleration options described above.

IDENTIFIER

Set this keyword to a named IDL variable that will contain the iToolID for the created tool. This value can then be used to reference this tool during overplotting operations or command-line-based tool management operations.

INTERPOLATE

Set this keyword to indicate that trilinear interpolation is to be used to determine the data value for each step on a ray. Setting this keyword improves the quality of images produced, at the cost of more computing time. especially when the volume has low resolution with respect to the size of the viewing plane. Nearest neighbor sampling is used by default.

LIGHTING_MODEL

Set this keyword to use the current lighting model during rendering in conjunction with a local gradient evaluation.

Note

Only DIRECTIONAL light sources are honored by the volume object. Because normals must be computed for all voxels in a lighted view, enabling light sources increases the rendering time.

LOCATION

Set this keyword to a two-element vector of the form [*x*, *y*] to specify the location of the upper left-hand corner of the tool relative to the display screen, in device units.

NAME

Set this keyword to a string to specify the name for this particular tool. The name is used for tool-related display purposes only–as the root of the hierarchy shown in the Tool Browser, for example.

OPACITY_TABLE0

Set this keyword to a 256-element byte array to specify an opacity table for Vol_0 if Vol_0 or Vol_0 and Vol_1 are present. If all the volume arguments are present, this keyword represents the opacity of the resulting RGBA volume. A value of 0 indicates complete transparency and a value of 255 indicates complete opacity. The default table is a linear ramp.

OPACITY_TABLE1

Set this keyword to a 256-element byte array to specify an opacity table for Vol_1 when Vol_0 and Vol_1 are present. A value of 0 indicates complete transparency and a value of 255 indicates complete opacity. The default table is a linear ramp.

OVERPLOT

Set this keyword to an iToolID to direct the graphical output of the particular tool to the tool specified by the provided iToolID.

Set this keyword to 1 (one) to place the graphical output for the command in the current tool. If no current tool exists, a new tool is created.

RENDER_EXTENTS

Set this keyword to draw a boundary around the rendered volume. The default (RENDER_EXTENTS = 2) is to draw a translucent boundary box. Possible values for this keyword are:

- 0 = Do not draw anything around the volume.
- 1 = Draw a wireframe around the volume.
- 2 = Draw a translucent box around the volume

RENDER_STEP

Set this keyword to a three element vector of the form [*x*, *y*, *z*] to specify the stepping factor through the voxel matrix. This keyword is only valid if render quality is set to high (RENDER_QUALITY = 2). The default render step is [1, 1, 1].

RENDER_QUALITY

Set this keyword to determine the quality of the rendered volume. The default (RENDER_QUALITY = 1) is low quality. Possible values for this keyword are:

- 1 = Low - Renders volume with a stack of two-dimensional texture maps.
- 2 = High - Use ray-casting rendering, see the COMPOSITE_FUNCTION for more details.

RGB_TABLE0

Set this keyword to a 3 by 256 or 256 by 3 byte array of RGB color values to specify a color table for Vol_0 if Vol_0 or Vol_0 and Vol_1 are present. If all the arguments are present, this keyword represents the RGB color values of all of these volumes. The default is a linear ramp

RGB_TABLE1

Set this keyword to a 3 by 256 or 256 by 3 byte array of RGB color values to specify a color table for Vol_1 when Vol_0 and Vol_1 are present. The default is a linear ramp.

SUBVOLUME

Set this keyword to a six-element vector of the form [x_{min}, y_{min}, z_{min}, x_{max}, y_{max}, z_{max}], which represents the sub-volume to be rendered. This keyword is the same as the BOUNDS keyword.

TITLE

Set this keyword to a string to specify the title for this particular tool. The title is displayed in the title bar of the tool.

TWO_SIDED

Set this keyword to force the lighting model to use a two-sided voxel gradient. The two-sided gradient is different from the one-sided gradient (default) in that the absolute value of the inner product of the light direction and the surface gradient is used instead of clamping to 0.0 for negative values.

VIEW_GRID

Set this keyword to a two-element vector of the form [*columns*, *rows*] to specify the view layout within the new tool. This keyword is only used if a new tool is being created (for example, if OVERPLOT, VIEW_NEXT, or VIEW_NUMBER are specified then VIEW_GRID is ignored).

VIEW_NEXT

Set this keyword to change the view selection to the next view following the currently-selected view before issuing any graphical commands. If the currently-selected view is the last one in the layout, then /VIEW_NEXT will cause the first view in the layout to become selected. This keyword is ignored if no current tool exists.

Note

The contents of the newly-selected view will be emptied unless /OVERPLOT is set.

VIEW_NUMBER

Set this keyword to change the currently-selected view to the view specified by the VIEW_NUMBER before issuing any graphical commands. The view number starts at 1, and corresponds to the position of the view within the graphics container (not necessarily the position on the screen). This keyword is ignored if no current tool exists.

Note

The contents of the newly-selected view will be emptied unless /OVERPLOT is set.

VOLUME_DIMENSIONS

A 3-element vector specifying the volume dimensions in terms of user data units. For example, specifying [0.1, 0.1, 0.1] would cause the volume to be rendered into a region that is 0.1 data units long on each side of the volume cube. If this parameter is not specified, the volume is rendered into a region the same size as the number of samples, with an origin of [0, 0, 0]. In this case, a volume with sample size of [20, 25, 20] would render into the region [0:19, 0:24, 0:19] in user data units. Use the VOLUME_LOCATION keyword to specify a different origin.

VOLUME_LOCATION

A 3-element vector specifying the volume location in user data units. Use this keyword to render the volume so that the first sample voxel appears at the specified location, instead of at [0, 0, 0], the default. Specify the location in terms of coordinates after the application of the VOLUME_DIMENSIONS values. For example, if the value of the VOLUME_DIMENSIONS keyword is [0.1, 0.1, 0.1] and you want the volume to be centered at the origin, set the VOLUME_LOCATION keyword to [-0.05, -0.05, -0.05].

[XYZ]MAJOR

Set this keyword to an integer representing the number of major tick marks. The default is -1, specifying that IDL will compute the number of tickmarks. Setting MAJOR equal to zero suppresses major tickmarks entirely.

[XYZ]MINOR

Set this keyword to an integer representing the number of minor tick marks. The default is -1, specifying that IDL will compute the number of tickmarks. Setting MINOR equal to zero suppresses minor tickmarks entirely.

[XYZ]RANGE

Set this keyword to the desired data range of the axis, a 2-element vector. The first element is the axis minimum, and the second is the maximum.

[XYZ]SUBTICKLEN

Set this keyword to a floating-point scale ratio specifying the length of minor tick marks relative to the length of major tick marks. The default is 0.5, specifying that the minor tick mark is one-half the length of the major tick mark.

[XYZ]TEXT_COLOR

Set this keyword to an RGB value specifying the color for the axis text. The default value is [0, 0, 0] (black).

[XYZ]TICKFONT_INDEX

Set this keyword equal to one of the following integers, which represent the type of font to be used for the axis text:

- 0 = Helvetica
- 1 = Courier
- 2 = Times
- 3 = Symbol
- 4 = Hershey

[XYZ]TICKFONT_SIZE

Set this keyword to an integer representing the point size of the font used for the axis text. The default is 12.0 points.

[XYZ]TICKFONT_STYLE

Set this keyword equal to one of the following integers, which represent the style of font to be used for the axis text:

- 0 = Normal
- 1 = Bold
- 2 = Italic
- 3 = Bold Italic

[XYZ]TICKFORMAT

Set this keyword to a string, or an array of strings, in which each string represents a format string or the name of a function to be used to format the tick mark labels. If an array is provided, each string corresponds to a level of the axis. The TICKUNITS keyword determines the number of levels for an axis.

If the string begins with an open parenthesis, it is treated as a standard format string. See “Format Codes” in Chapter 10 of the *Building IDL Applications* manual.

If the string does not begin with an open parenthesis, it is interpreted as the name of a callback function to be used to generate tick mark labels.

If TICKUNITS are not specified:

- The callback function is called with three parameters: *Axis*, *Index*, and *Value*, where:
- *Axis* is the axis number: 0 for X axis, 1 for Y axis, 2 for Z axis
- *Index* is the tick mark index (indices start at 0)
- *Value* is the data value at the tick mark (a double-precision floating point value)

If TICKUNITS are specified:

The callback function is called with four parameters: *Axis*, *Index*, *Value*, and *Level*, where:

- *Axis*, *Index*, and *Value* are the same as described above.
- *Level* is the index of the axis level for the current tick value to be labeled. (Level indices start at 0.)

Used with the LABEL_DATE function, this property can easily create axes with date/time labels.

[XYZ]TICKINTERVAL

Set this keyword to a floating-point scalar indicating the interval between major tick marks for the first axis level. The default value is computed according to the axis [XYZ]RANGE and the number of major tick marks ([XYZ]MAJOR). The value of this keyword takes precedence over the value set for the [XYZ]MAJOR keyword.

For example, if TICKUNITS = ['S', 'H', 'D'], and TICKINTERVAL = 30, then the interval between major ticks for the first axis level will be 30 seconds.

[XYZ]TICKLAYOUT

Set this keyword to integer scalar that indicates the tick layout style to be used to draw each level of the axis.

Valid values include:

- 0 = The axis line, major tick marks and tick labels are all included. Minor tick marks only appear on the first level of the axis. This is the default tick layout style.
- 1 = Only the labels for the major tick marks are drawn. The axis line, major tick marks, and minor tick marks are omitted.
- 2 = Each major tick interval is outlined by a box. The tick labels are positioned within that box (left-aligned). For the first axis level only, the major and minor tick marks will also be drawn.

Note

For all tick layout styles, at least one tick label will appear on each level of the axis (even if no major tick marks fall along the axis line). If there are no major tick marks, the single tick label will be centered along the axis.

[XYZ]TICKLEN

Set this keyword to a floating-point value that specifies the length of each major tick mark, measured in data units. The recommended, and default, tick mark length is 0.2. IDL converts, maintains, and returns this data as double-precision floating-point.

[XYZ]TICKNAME

Set this keyword to a string array of up to 30 elements that controls the annotation of each tick mark.

[XYZ]TICKUNITS

Set this keyword to a string (or a vector of strings) indicating the units to be used for axis tick labeling. If more than one unit is provided, the axis will be drawn in multiple levels, one level per unit.

The order in which the strings appear in the vector determines the order in which the corresponding unit levels will be drawn. The first string corresponds to the first level (the level nearest to the primary axis line).

Valid unit strings include:

- "Numeric"
- "Years"
- "Months"
- "Days"
- "Hours"
- "Minutes"
- "Seconds"
- "Time" - Use this value to indicate that the tick values are time values; IDL will determine the appropriate time intervals and tick label formats based upon the range of values covered by the axis.
- ""- Use the empty string to indicate that no tick units are being explicitly set. This implies that a single axis level will be drawn using the "Numeric" unit. This is the default setting.

If any of the time units are utilized, then the tick values are interpreted as Julian date/time values. Note that the singular form of each of the time value strings is also acceptable (e.g, TICKUNITS = 'Day' is equivalent to TICKUNITS = 'Days').

Note

Julian values must be in the range -1095 to 1827933925, which corresponds to calendar dates 1 Jan 4716 B.C.E. and 31 Dec 5000000hidd, respectively.

[XYZ]TICKVALUES

Set this keyword to a floating-point vector of data values representing the values at each tick mark. If TICKVALUES is set to 0, the default, IDL computes the tick values based on the axis range and the number of major ticks. IDL converts, maintains, and returns this data as double-precision floating-point.

[XYZ]TITLE

Set this keyword to a string representing the title of the specified axis.

ZBUFFER

Set this keyword to clip the rendering to the current Z-buffer and then update the buffer.

ZERO_OPACITY_SKIP

Set this keyword to skip voxels with an opacity of 0. This keyword can increase the output contrast of MIP (MAXIMUM_INTENSITY) projections by allowing the background to show through. If this keyword is set, voxels with an opacity of zero will not modify the Z-buffer. The default (not setting the keyword) continues to render voxels with an opacity of zero.

Examples

In the IDL Intelligent Tools system, data can be imported from the IDL Command Line (as described in Example 1), or data can be imported via the **File** menu in the iTool window (as described in Examples 2 and 3). For detailed information on importing data via the iTool file menu, refer to "Data Import Methods" in Chapter 2 of the *iTool User's Guide* manual.

Example 1

This example shows how to use the IDL Command Line to bring data into the iVolume tool.

At the IDL Command Line, enter:

```
file = FILEPATH('clouds3d.dat', $
   SUBDIRECTORY = ['examples', 'data'])
RESTORE, file
IVOLUME, clouds
```

Derive an interval volume by selecting **Operations** → **Volume** → **Interval Volume**. In the Interval Volume Value Selector dialog, change the minimum value to `0.2` and the **Decimate: % of original surface** slider to `20`, then click **OK**.

The following figure displays the output of this example:

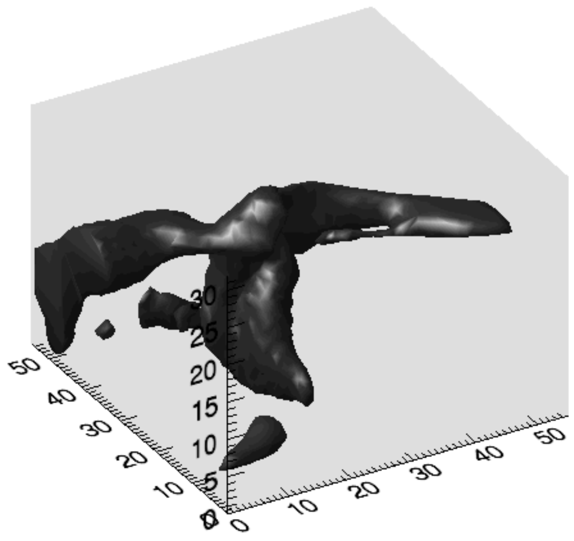


Figure 3-14: Cloud Interval Volume iVolume Example

Example 2

This example shows how to use the iTool **File** → **Open** command to load binary data into the iVolume tool.

At the IDL Command Line, enter:

```
IVOLUME
```

Select **File** → **Open** to display the Open dialog, then browse to find `head.dat` in the `examples/data` directory in the IDL distribution, and click **Open**.

In the Binary Template dialog, click **New Field**, and enter the following information in the New Field dialog:

- **Field Name:** `data` (or a name of your choosing)
- **Type:** `Byte (unsigned 8-bits)`
- **Number of Dimensions:** `3`
- **1st Dimension Size:** `80`

- **2nd Dimension Size:** `100`
- **3rd Dimension Size:** `57`

Click **OK** to close the New Field dialog and the Binary Template dialog, and the image is displayed.

Note

For more information on using the Binary Template to import data, see "Using the BINARY_TEMPLATE Function" in Chapter 15 of the *Using IDL* manual.

Select **Operations** → **Volume** → **Isosurface**, and insert an isosurface with a value of 60, decimated to 20% of the original surface.

The following figure displays the output of this example:

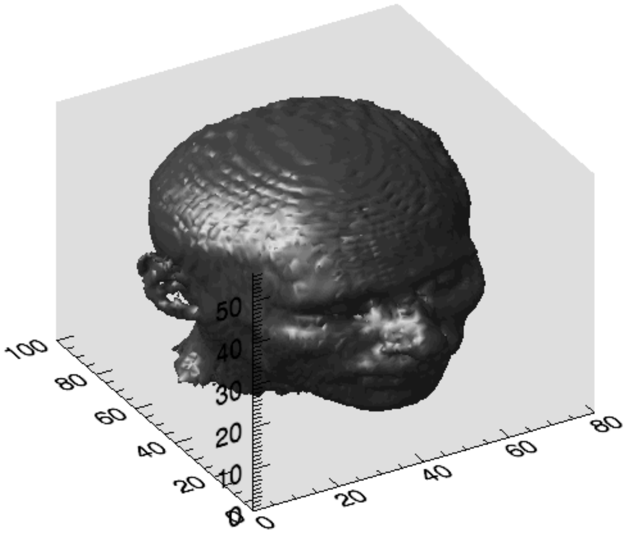


Figure 3-15: Human Head MRI Isosurface iVolume Example

Example 3

This example shows how to use the **File** → **Import** command to load binary data into the iVolume tool.

At the IDL Command Line, enter:

```
IVOLUME
```

Select **File** → **Import** to display the IDL Import Data wizard.

1. At Step 1, select **From a File** and click **Next**>>.
2. At Step 2, under **File Name:**, browse to find `jet.dat` in the `examples/data` directory in the IDL distribution, and click **Next**>>.
3. At Step 3, select **Volume** and click **Finish**.

The Binary Template wizard is displayed. In the Binary Template, change **File's byte ordering** to `Little Endian`. Then, click **New Field**, and enter the following information in the New Field dialog:

- **Field Name:** `data` (or a name of your choosing)
- **Type:** `Byte (unsigned 8-bits)`
- **Number of Dimensions:** `3`
- **1st Dimension Size:** `81`
- **2nd Dimension Size:** `40`
- **3rd Dimension Size:** `101`

Click **OK** to close the New Field dialog and the Binary Template dialog, and the volume is displayed.

Select **Operations** → **Volume** → **Image Plane** to display a plane in the *x*-direction. Double-click on the plane to access its properties through the property sheet. Change the **Orientation** setting to `Z`. You can drag the image to see it at different z values by clicking on the edge of the image plane.

The following figure displays the output of this example:

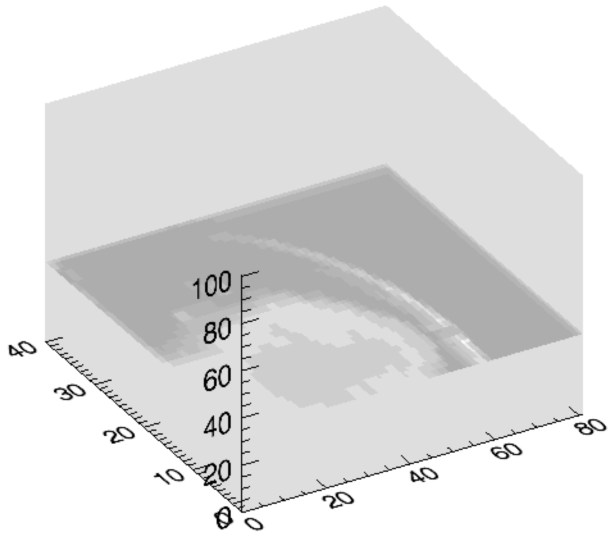


Figure 3-16: Plasma Jet Image Plane iVolume Example

Example 4

This example shows how to use a second volume argument to cut away a section of the first volume argument.

First, load the MRI head data into IDL. At the IDL Command Line, enter:

```
file = FILEPATH('head.dat', SUBDIRECTORY = ['examples', 'data'])
data0 = READ_BINARY(file, DATA_DIMS = [80, 100, 57])
```

Then, create the second volume that will cut away the upper left corner of the head. At the IDL Command Line, enter:

```
data1 = BYTARR(80, 100, 57) + 1B
data1[0:39, *, 28:56] = 0B
```

Derive the color and opacity tables for the second volume. At the IDL Command Line, enter:

```
rgbTable1 = [[BYTARR(256)], [BYTARR(256)], [BYTARR(256)]]
rgbTable1[1, *] = [255, 255, 255]
opacityTable1 = BYTARR(256)
opacityTable1[1] = 255
```

Now, display the two volumes. At the IDL Command Line, enter:

```
IVOLUME, data0, data1, RGB_TABLE1 = rgbTable1, $
   OPACITY_TABLE1 = opacityTable1, /AUTO_RENDER
```

The following figure displays the output of this example:

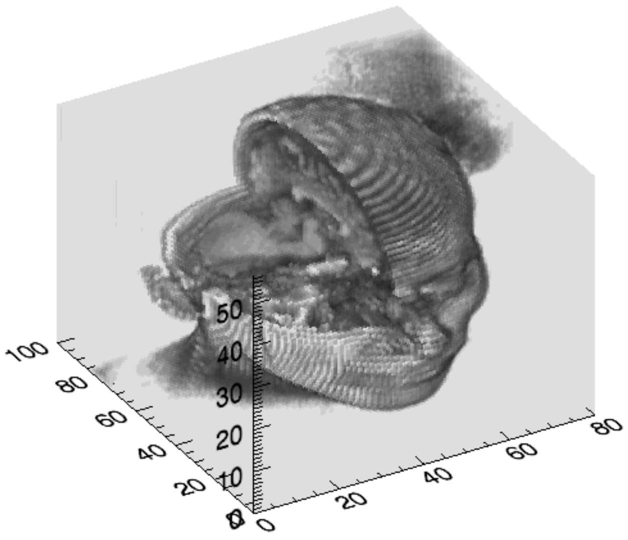


Figure 3-17: Cut Away iVolume Example

Example 5

This example shows how to use all the volume arguments to display an RGB (Red, Green, Blue) volume.

First, create the volumes to contain primary colors (black, red, green, blue, yellow, cyan, magenta, and white) in each corner. At the IDL Command Line, enter:

```
vol0 = BYTARR(32, 32, 32)
vol1 = BYTARR(32, 32, 32)
vol2 = BYTARR(32, 32, 32)
vol3 = BYTARR(32, 32, 32)
vol0[0:15, *, *] = 255
vol1[*, 0:15, *] = 255
vol2[*, *, 0:15] = 255
vol3[*, *, *] = 128
```

Then, derive the color and opacity tables. At the IDL Command Line, enter:

```
rgbTable = [[BYTARR(256)], [BYTARR(256)], [BYTARR(256)]]
opacityTable = BINDGEN(256)
```

Now, display the two volumes. At the IDL Command Line, enter:

```
IVOLUME, vol0, vol1, vol2, vol3, RGB_TABLE0 = rgbTable, $
   OPACITY_TABLE0 = opacityTable, /AUTO_RENDER
```

The following figure displays the output of this example:

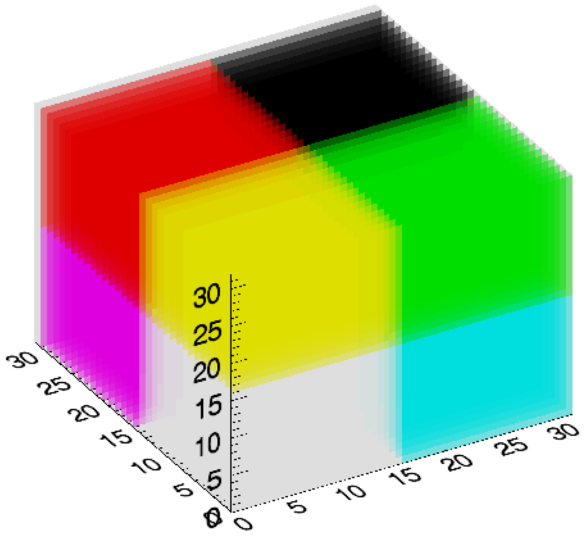


Figure 3-18: RGB iVolume Example

Note

The white corner of this example volume is actually gray to distinguish it from the white background.

Version History

Introduced: 6.0

LOGICAL_AND

The LOGICAL_AND function performs a logical AND operation on its arguments. It returns True (1) if both of its arguments are non-zero (non-NULL for strings and heap variables), or False (0) otherwise.

The LOGICAL_AND function is similar to the AND operator, except that it performs a logical “and” rather than a bitwise “and” on its arguments.

Note

LOGICAL_AND always returns either 0 or 1, unlike the AND operator, which performs a bitwise AND operation on integers, and returns one of the two arguments for other types.

Unlike the && operator, LOGICAL_AND accepts multi-element arrays as its arguments. In addition, where the && operator *short-circuits* if it can determine the result by evaluating only the first argument, all arguments to a function are always evaluated.

Syntax

Result = LOGICAL_AND(*Arg1*, *Arg2*)

Return Value

Integer zero (false) or one (true) if both arguments are scalars, or an array of zeroes and ones if either argument is an array.

Arguments

Arg1, Arg2

The expressions on which the logical AND operation is to be carried out. The arguments can be scalars or arrays of any type other than structure.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL’s *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See Appendix D, “Thread Pool Keywords” for details.

Example

At the IDL Command line, enter:

```
PRINT, LOGICAL_AND(2,4), LOGICAL_AND(2,0), LOGICAL_AND(0,4), $
   LOGICAL_AND(0,0)
```

IDL Prints:

```
1   0   0   0
```

Version History

Introduced: 6.0

See Also

“Logical Operators” in the *Building IDL Applications* manual, “Bitwise Operators” in the *Building IDL Applications* manual, LOGICAL_OR, LOGICAL_TRUE

LOGICAL_OR

The LOGICAL_OR function performs a logical OR operation on its arguments. It returns True (1) if either of its arguments are non-zero (non-NULL for strings and heap variables), and False (0) otherwise.

The LOGICAL_OR function is similar to the OR operator, except that it performs a logical “or” rather than a bitwise “or” on its arguments.

Note

LOGICAL_OR always returns either 0 or 1, unlike the OR operator, which performs a bitwise OR operation on integers, and returns one of the two arguments for other types.

Unlike the `||` operator, LOGICAL_OR accepts multi-element arrays as its arguments. In addition, where the `||` operator *short-circuits* if it can determine the result by evaluating only the first argument, all arguments to a function are always evaluated.

Syntax

Result = LOGICAL_OR(*Arg1*, *Arg2*)

Return Value

Integer zero (false) or one (true) if both operands are scalars, or an array of zeroes and ones if either operand is an array.

Arguments

Arg1, Arg2

The expressions on which the logical OR operation is to be carried out. The arguments can be scalars or arrays of any type other than structure.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL’s *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See Appendix D, “Thread Pool Keywords” for details.

Example

At the IDL Command Line, enter:

```
PRINT, LOGICAL_OR(2,4), LOGICAL_OR(2,0), LOGICAL_OR(0,4), $
   LOGICAL_OR(0,0)
```

IDL Prints:

```
1   1   1   0
```

Version History

Introduced: 6.0

See Also

“Logical Operators” in the *Building IDL Applications* manual, “Bitwise Operators” in the *Building IDL Applications* manual, LOGICAL_AND, LOGICAL_TRUE

LOGICAL_TRUE

The LOGICAL_TRUE function returns True (1) if its arguments are non-zero (non-NULL for strings and heap variables), and False (0) otherwise.

Note

For a given argument, the value returned by LOGICAL_TRUE is the opposite of the value returned by the ~ operator.

Syntax

Result = LOGICAL_TRUE(*Arg*)

Return Value

Integer zero (false) or one (true) if the argument is a scalar, or an array of zeroes and ones if the argument is an array.

Arguments

Arg

The expression on which the logical truth evaluation is to be carried out. The argument can be a scalar or an array of any type other than structure.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See Appendix D, "Thread Pool Keywords" for details.

Example

At the IDL Command Line, enter:

```
PRINT, LOGICAL_TRUE(2), LOGICAL_TRUE(0)
```

IDL Prints:

```
1   0
```

Version History

Introduced: 6.0

See Also

“Logical Operators” in the *Building IDL Applications* manual, “Bitwise Operators” in the *Building IDL Applications* manual, KEYWORD_SET, LOGICAL_AND, LOGICAL_OR

PATH_CACHE

The PATH_CACHE procedure is used to control IDL's use of the *path cache*. By default, as IDL searches directories included in the !PATH system variable for `.pro` or `.sav` files to compile, it creates an in-memory list of *all* `.pro` and `.sav` files contained in each directory. When IDL later searches for a `.pro` or `.sav` file, before attempting to open the file in a given directory, IDL checks the path cache to determine whether the directory has already been cached. If the directory is included in the cache, IDL uses the cached information to determine whether the file will be found in that directory, and will only attempt to open the file there if the cache tells it that the file exists. By eliminating unnecessary attempts to open files, the path cache speeds the path searching process.

The path cache is enabled by default, and in almost all cases its operation is transparent to the IDL user, save for the boost in path searching speed it provides. Because the cache automatically adjusts to changes made to IDL's path, use of PATH_CACHE should not be necessary in typical IDL operation. It is provided to allow complete control over the details of how and when the caching operation is performed.

- For information on when the path cache is *not* used, see "Situations in which IDL will not use the Path Cache" on page 253.
- For information on disabling the path cache, see "Disabling the Path Cache" on page 254.

Note

Prior to IDL 6.0, IDL did not use a path cache. Aside from the improvement in performance, the behavior of IDL with the path cache is identical to that without in almost all cases. The rare cases in which it differs, and options for disabling its use, are discussed in "Options for Avoiding Use of the Path Cache" on page 255.

About the Path Cache

The first time an IDL session attempts to call a function or procedure written in the IDL language, it must locate and compile the file containing the code for that routine. The file containing the routine must have the same name as the routine, with either a `.pro` or a `.sav` extension. After trying to open the file in the user's current working directory, IDL will attempt to open the file in each of the directories listed in the !PATH system variable, in the order specified by !PATH. The search stops when a file with the desired name is found or no directories remain in !PATH.

By default, IDL maintains an in-memory cache of the locations of `.pro` and `.sav` files stored in directories included in the !PATH system variable. The path cache is built automatically during normal operation, as IDL searches the directories specified by !PATH. Once a directory is cached, IDL knows whether or not it contains a given file, without the need to actually attempt to open that file. This information allows IDL to bypass directories that do not contain the desired file, providing a significant boost in the speed of path searching. The path cache can significantly improve the startup speed of large, object-oriented applications, because method resolution requires extensive path searching.

The path cache operates on a per-directory basis; if IDL searches a directory for a `.pro` or `.sav` file, the locations of *all* `.pro` and `.sav` files in that directory are added to the cache, and the directory is not searched again until the cache is cleared and rebuilt.

Note

The current contents of the path cache can be viewed using the PATH_CACHE keyword to the HELP procedure.

Syntax

PATH_CACHE[, /CLEAR] [, /ENABLE] [, /REBUILD]

Arguments

None.

Keywords

CLEAR

Set this keyword to clear the entire contents of the path cache, leaving it completely empty. If path caching is enabled, IDL will begin rebuilding the cache the next time it needs to locate a `.pro` or `.sav` file. If you wish to prevent the rebuilding of the cache, set the ENABLE keyword equal to zero as well.

Note

The .RESET_SESSION executive command clears the entire path cache as part of resetting the IDL session.

ENABLE

Set this keyword to a non-zero value to specify that IDL should use the path cache when searching for files and also add new directories to the cache as they are opened. Set this keyword to zero to disable use of the cache when searching for files, and to discontinue adding new directories.

Note

Disabling the cache does not cause the current contents of the cache to be discarded. To discard the cache information, specify the CLEAR keyword.

REBUILD

Set this keyword to discard the current contents of the path cache (as if the CLEAR keyword had been specified), and then immediately rebuild the cache by searching the directories specified by the current value of the !PATH system variable for `.pro` and `.sav` files.

Note

If !PATH contains many directories, or if access to those directories is slow, rebuilding the cache using this method may also be slow. In many cases, the CLEAR keyword is sufficient, since IDL will rebuild the empty cache as program execution requires it to search for `.pro` and `.sav` files.

Situations in which IDL will not use the Path Cache

By default, IDL uses the path cache whenever it tries to locate `.pro` or `.sav` files. However, IDL will never use the path cache in the following situations:

Current Working Directory

The path cache is neither checked nor added to if the file being searched for exists in the current working directory. Before IDL searches !PATH for a file to compile, it always looks in the current working directory without checking the cache.

Relative Paths

The path cache does not cache directories specified relative to the current directory, even though relative paths are allowed in the specification of !PATH.

An absolute (or *fully qualified*) path is a path that completely specifies the location of a file. Under UNIX, an absolute path is specified relative to the root of the filesystem, and therefore starts with a slash (`/`) character. Under Microsoft Windows, an absolute path starts with a drive letter (`C:`, for example) or a double backslash (`\\`) (if the file is specified using the Universal Naming Convention format). In contrast, a relative path is incomplete, and must be interpreted relative to the current working directory of the IDL process. IDL only caches absolute paths.

Executive Commands

The path cache is neither checked nor added to when a `.COMPILE` or `.RUN` executive command is issued. In such cases, IDL performs a standard directory-by-directory search of the directories included in !PATH.

IDL_NOCACHE File Present

IDL will not cache the contents of any directory that contains a file named `IDL_NOCACHE`. See "Marking Specific Directories as Uncacheable" on page 255 for additional information on this feature.

Path Cache Disabled

IDL will neither check nor add files to the path cache if it has been disabled. See "Disabling the Path Cache", below, for additional information.

Disabling the Path Cache

By default, IDL caches the locations of `.pro` and `.sav` files in all directories specified by the !PATH system variable. Use of the path cache can be fully disabled in the following ways:

1. By issuing the PATH_CACHE command with the ENABLE keyword set equal to zero. This will disable the path cache until you manually re-enable it, or for the duration of the current IDL session. See the description of the ENABLE keyword, above, for details.
2. By unchecking the "Enable Path Caching" checkbox on the Path tab of the IDLDE Preferences dialog. See "Path Preferences" in Chapter 5 of the *Using IDL* manual for details.
3. By defining an environment variable named IDL_PATH_CACHE_DISABLE before starting IDL. See "Environment Variables Used by IDL" in Chapter 1 of the *Using IDL* manual for details.

In addition, you can selectively disable use of the path cache for specific directories by creating a file named `IDL_NOCACHE` in the directory. See "Marking Specific Directories as Uncacheable", below, for details.

Marking Specific Directories as Uncacheable

You can mark specific directories as being uncacheable even though the directory is included in !PATH. To do so, create a file named `IDL_NOCACHE` in that directory.

Note

IDL does not inspect the contents of an `IDL_NOCACHE` file; it can contain anything you wish, or nothing at all. Under Unix operating systems, the `IDL_NOCACHE` file must be named exactly as shown, using all uppercase characters in the name. Under Microsoft Windows, the characters can have any case, but RSI suggests you use upper case for consistency.

When IDL encounters a directory containing an `IDL_NOCACHE` file during normal path searching, it makes a special entry in the path cache telling it that the directory must not be cached. Once this is done, all future attempts to locate files in that directory will be done without using cached information.

Note

If the directory to which you add an `IDL_NOCACHE` file has already been added to the path cache for the current IDL session, you must clear the existing cache (using the CLEAR keyword to the PATH_CACHE procedure) before the no-cache setting will take effect.

To re-enable path caching for a directory that has been marked as uncacheable, remove the `IDL_NOCACHE` file, and then reset IDL's path cache in one of the following ways:

- Specify the CLEAR keyword to the PATH_CACHE procedure.
- Issue the .RESET_SESSION executive command.
- Exit and restart the IDL session.

Options for Avoiding Use of the Path Cache

In most cases, the files contained in directories included in !PATH do not change during an IDL session. In such cases the path cache is completely transparent to the IDL user, and serves only to speed compilation of IDL routines. As a result, there is rarely a reason to globally disable the path cache.

If files are created or deleted in a directory included in !PATH during an IDL session, the path cache can become confused and provide bad information to IDL about the contents of that directory. There are several ways to handle this situation. The following list of alternatives is given in rough order of preference, with the easiest and lowest-impact options given first:

1. Leave the path cache enabled, and change your current working directory to the directory in which files are created or deleted. Since IDL checks the current working directory before checking the directories in !PATH, use of the path cache does not affect IDL's ability to find these files.
2. If the addition or deletion of files in a directory included in !PATH is a rare occurrence, leave the path cache enabled and clear it in one of the following ways after the contents of the directory have changed:
   - Specify the CLEAR keyword to the PATH_CACHE procedure.
   - Issue the .RESET_SESSION executive command.
   - Exit and restart the IDL session.
3. Leave the path cache enabled and use the .COMPILE or .RUN executive commands to force the compilation of any file, regardless of the contents of the path cache.
4. If you have a directory (other than your current working directory) in which files are regularly added or deleted during the execution of IDL sessions, you can leave path caching enabled but explicitly disable caching of that specific directory by creating an `IDL_NOCACHE` file, as described in "Marking Specific Directories as Uncacheable" on page 255. This approach works for all IDL sessions that access the directory, and is therefore convenient in long-term or multi-user situations.
5. You can completely disable operation of the path cache using one of the methods described under "Disabling the Path Cache" on page 254. This is not recommended, because most directories are not dynamic, and completely disabling path caching sacrifices the performance advantages of caching directories whose contents *are* static.

Note on Behavior at Startup

Depending on the value of your !PATH system variable, you may notice that some directories are being cached immediately when IDL starts up. This will occur if your path definition string includes the `<IDL_DEFAULT>` token, or if one or more entries include the "+" symbol. In these cases, in order for IDL to build the !PATH system variable, it must inspect subdirectories of the specified directories for the presence of `.pro` and `.sav` files, with the side effect of adding these directories to the path cache. See EXPAND_PATH for a discussion of IDL's path expansion behavior.

Examples

The following statement disables path caching for the current session:

```
PATH_CACHE, ENABLE = 0
```

The following statement disables path caching for the current session and throws away the current contents of the cache:

```
PATH_CACHE, ENABLE = 0, /CLEAR
```

Suppose you want to remove a directory included in !PATH from the cache without resetting your IDL session. The following statements cause the specified directory not to be included in future caching by creating a file named `IDL_NOCACHE` in that directory:

```
OPENW, UNIT = u, '/home/idluser/idl_dev_dir/IDL_NOCACHE', /GET_LUN
FREE_LUN, u
```

The OPENW and FREE_LUN statements create an empty file with the desired name in the target directory. Executing the following statement clears the cache so as to reflect the change in the current IDL session:

```
PATH_CACHE, /CLEAR
```

The next time IDL encounters this directory in a path search, it will see the presence of the `IDL_NOCACHE` and make a note in the path cache that the directory is not cacheable.

Note

You can also create the `IDL_NOCACHE` file outside IDL using any convenient command (text editor, Unix `touch` command, *etc*.). If the file is created outside IDL, only the `PATH_CACHE, /CLEAR` statement is necessary.

Version History

Introduced: 6.0

See Also

.FULL_RESET_SESSION, .RESET_SESSION, “!PATH” in Appendix D, “Environment Variables Used by IDL” in Chapter 1 of the *Using IDL* manual, “Path Preferences” in Chapter 5 of the *Using IDL* manual

WIDGET_PROPERTYSHEET

The WIDGET_PROPERTYSHEET function creates a property sheet widget, which exposes the *properties* of an IDL object subclassed from the IDLitComponent class in a graphical interface. The property sheet widget must be a child of a base or tab widget, and it cannot be the parent of any other widget.

The property sheet widget exposes the properties of an IDL object that subclasses from the IDLitComponent class, which was designed for use by the IDL iTools system. As a result, all IDLit* objects subclass from IDLitComponent, so properties of object classes written for the IDL iTools system can be displayed in a property sheet. In addition, all IDLgr* objects subclass from IDLitComponent, which means that properties of standard IDL graphics objects can be displayed in a property sheet even if the rest of the iTools framework is not in use.

In order to be shown in a property sheet, object properties must be *registered* and *visible*. In addition, in order for property values shown in a property sheet to be editable by the user, the property must be *sensitive*. For information on registering properties, see “Registering Properties” in Chapter 4 of the *iTool Developer’s Guide* manual. For information on making properties visible and sensitive, see “Property Attributes” in Chapter 4 of the *iTool Developer’s Guide* manual.

Syntax

Result = WIDGET_PROPERTYSHEET(*Parent* [, /ALIGN_BOTTOM |, /ALIGN_CENTER |, /ALIGN_LEFT |, /ALIGN_RIGHT |, /ALIGN_TOP] [, /CONTEXT_EVENTS] [, EVENT_FUNC=*string*] [, EVENT_PRO=*string*] [, FONT=*string*] [, FUNC_GET_VALUE=*string*] [, KILL_NOTIFY=*string*] [, /NO_COPY] [, NOTIFY_REALIZE=*string*] [, PRO_SET_VALUE=*string*] [, SCR_XSIZE=*width*] [, SCR_YSIZE=*height*] [, /SENSITIVE] [, /TRACKING_EVENTS] [, UNAME=*string*] [,UNITS={0 | 1 | 2}] [, UVALUE=*value*] [, VALUE=*value*] [, XOFFSET=*value*] [, XSIZE=*value*] [, YOFFSET=*value*] [, YSIZE=*value*])

Return Value

The returned value of this function is the widget ID of the newly-created property sheet widget.

Arguments

Parent

The widget ID of the parent for the new property sheet widget. *Parent* must be a base or tab widget.

Keywords

ALIGN_BOTTOM

Set this keyword to align the new widget with the bottom of its parent base. To take effect, the parent must be a ROW base.

ALIGN_CENTER

Set this keyword to align the new widget with the center of its parent base. To take effect, the parent must be a ROW or COLUMN base. In ROW bases, the new widget will be vertically centered. In COLUMN bases, the new widget will be horizontally centered.

ALIGN_LEFT

Set this keyword to align the new widget with the left side of its parent base. To take effect, the parent must be a COLUMN base.

ALIGN_RIGHT

Set this keyword to align the new widget with the right side of its parent base. To take effect, the parent must be a COLUMN base.

ALIGN_TOP

Set this keyword to align the new widget with the top of its parent base. To take effect, the parent must be a ROW base.

CONTEXT_EVENTS

Set this keyword to cause *context menu events* (or simply *context events*) to be issued when the user clicks the right mouse button over the widget. Set the keyword to 0 (zero) to disable such events. Context events are intended for use with context-sensitive menus (also known as pop-up or shortcut menus); pass the context event ID to the WIDGET_DISPLAYCONTEXTMENU procedure within your widget program's event handler to display the context menu.

For more on detecting and handling context menu events, see “Context-Sensitive Menus” in Chapter 26 of the *Building IDL Applications* manual.

Note

With regard to /CONTEXT_EVENTS, the Motif and Windows version of the property sheet differ very slightly. In the Motif version, individually desensitized cells cannot generate context events, though their row label can.

EVENT_FUNC

A string containing the name of a function to be called by the WIDGET_EVENT function when an event arrives from a widget in the widget hierarchy rooted at the newly-created widget.

EVENT_PRO

A string containing the name of a procedure to be called by the WIDGET_EVENT function when an event arrives from a widget in the widget hierarchy rooted at the newly-created widget.

FONT

The name of the font to be used by the widget. The font specified is a device font (an X Windows font on Motif systems; a TrueType or PostScript font on Windows systems). See “About Device Fonts” in Appendix I of the *IDL Reference Guide* manual for details on specifying names for device fonts. If this keyword is omitted, the default font is used.

Note

On Microsoft Windows platforms, if FONT is not specified, IDL uses the system default font. Different versions of Windows use different system default fonts; in general, the system default font is the font appropriate for the version of Windows in question.

FUNC_GET_VALUE

A string containing the name of a function to be called when the GET_VALUE keyword to the WIDGET_CONTROL procedure is called for this widget. Using this technique allows you to change the value that should be returned for a widget. Compound widgets use this ability to define their values transparently to the user.

KILL_NOTIFY

Set this keyword to a string that contains the name of a procedure to be called automatically when the specified widget dies. Each widget is allowed a single such “callback” procedure. It can be removed by setting the routine to the null string (`''`).

The callback routine is called with the widget identifier as its only argument. At that point, the widget identifier can only be used with the WIDGET_CONTROL procedure to get or set the user value. All other requests that require a widget ID are disallowed for the target widget. The callback is not issued until the WIDGET_EVENT function is called.

Note

A procedure specified via the CLEANUP keyword to XMANAGER will override a procedure specified for your application’s top-level base with WIDGET_BASE, KILL_NOTIFY.

NO_COPY

Usually, when setting or getting widget user values, either at widget creation or using the SET_UVALUE and GET_UVALUE keywords to WIDGET_CONTROL, IDL makes a second copy of the data being transferred. Although this technique is fine for small data, it can have a significant memory cost when the data being copied is large.

If the NO_COPY keyword is set, IDL handles these operations differently. Rather than copy the source data, it takes the data away from the source and attaches it directly to the destination. This feature can be used by compound widgets to obtain state information from a UVALUE without all the memory copying that would otherwise occur. However, it has the side effect of causing the source variable to become undefined. During a set operation (using the UVALUE keyword to WIDGET_BASE or the SET_UVALUE keyword to WIDGET_CONTROL), the variable passed as value becomes undefined. During a get operation (GET_UVALUE keyword to WIDGET_CONTROL), the user value of the widget in question becomes undefined.

NOTIFY_REALIZE

Set this keyword to a string that contains the name of a procedure to be called automatically when the specified widget is realized. This callback occurs just once (because widgets are realized only once). Each widget is allowed a single such callback procedure. It can be removed by setting the routine to the null string (`''`). The callback routine is called with the widget ID as its only argument.

PRO_SET_VALUE

A string containing the name of a procedure to be called when the SET_VALUE keyword to the WIDGET_CONTROL procedure is called for this widget. Using this technique allows you to designate a routine that sets the value for a widget. Compound widgets use this ability to define their values transparently

SCR_XSIZE

Set this keyword to the desired screen width of the widget, in units specified by the UNITS keyword (pixels are the default). In many cases, setting this keyword is the same as setting the XSIZE keyword.

SCR_YSIZE

Set this keyword to the desired screen height of the widget, in units specified by the UNITS keyword (pixels are the default). In many cases, setting this keyword is the same as setting the YSIZE keyword.

SENSITIVE

Set this keyword to control the initial sensitivity state of the widget.

If SENSITIVE is zero, the widget becomes insensitive. If nonzero, it becomes sensitive. When a widget is sensitive, it has normal appearance and can receive user input. For example, a sensitive button widget can be activated by moving the mouse cursor over it and pressing a mouse button. When a widget is insensitive, it indicates the fact by changing its appearance, looking disabled, and it ignores any input.

Sensitivity can be used to control when a user is allowed to manipulate the widget.

Note

Some widgets do not change their appearance when they are made insensitive, but they cease generating events.

After creating the widget hierarchy, you can change the sensitivity state using the SENSITIVE keyword with the WIDGET_CONTROL procedure.

TRACKING_EVENTS

Set this keyword to cause widget tracking events to be issued for the widget whenever the mouse pointer enters or leaves the region covered by that widget. For the structure of tracking events, see “TRACKING_EVENTS” in the *IDL Reference Guide* manual in the documentation for WIDGET_BASE.

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.

UNITS

Set UNITS equal to 0 (zero) to specify that all measurements are in pixels (this is the default), to 1 (one) to specify that all measurements are in inches, or to 2 (two) to specify that all measurements are in centimeters.

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.

The user value for a widget can be accessed and modified at any time by using the GET_UVALUE and SET_UVALUE keywords to the WIDGET_CONTROL procedure.

VALUE

Set this keyword to the object reference or array of object references to objects that subclass from the IDLitComponent class. Registered properties of the specified objects will be displayed in the property sheet.

If a single object reference is supplied, the property sheet will have a single column containing the object's properties. If an array of object references is supplied, the property sheet will have multiple columns.

Note

Due to limitations of the user interface controls that underlie the property sheet widget, a property sheet can display properties for at most 100 component objects.

Note

All object references must be to objects of the same type.

If no object references are supplied, the property sheet will initially be empty. Object references can be loaded into an existing property sheet using the SET_VALUE keyword to WIDGET_CONTROL.

XOFFSET

The horizontal offset of the widget in units specified by the UNITS keyword (pixels are the default) relative to its parent.

Specifying an offset relative to a row or column major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a bulletin board base widget. Note that it is best to avoid using this style of widget layout.

XSIZE

The desired width, in average character widths, for the widget's font, not including a possible vertical scrollbar and any frame thickness. If neither XSIZE nor SCR_XSIZE is specified, then the property sheet widget will use a default width. This default width is computed by adding the room needed for the property names to the width of a color cell.

YOFFSET

The vertical offset of the widget in units specified by the UNITS keyword (pixels are the default) relative to its parent. This offset is specified relative to the *upper left* corner of the parent widget.

Specifying an offset relative to a row or column major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a bulletin board base widget. Note that it is best to avoid using this style of widget layout.

YSIZE

The desired height of the widget, in number of visible properties. The ultimate height of the property sheet in pixels will include the heights of the column header, the possible horizontal scrollbar, and any frame. If neither YSIZE nor SCR_YSIZE is specified, the property sheet will use a default height. This default is based on the number of rows: 10, or the number of visible properties, whichever is less.

Keywords to WIDGET_CONTROL

A number of keywords to the WIDGET_CONTROL affect the behavior of property sheet widgets. In addition to those keywords that affect all widgets, the following keyword is particularly useful: REFRESH _PROPERTY.

Keywords to WIDGET_INFO

Some keywords to WIDGET_INFO return information that applies specifically to property sheet widgets. In addition to those keywords that apply to all widgets, the following keywords are particularly useful: COMPONENT, PROPERTY_VALID, PROPERTY_VALUE.

Widget Events Returned by Property Sheet Widgets

Several variations of the property sheet widget event structure depend upon the specific event being reported. All of these structures contain the standard three fields (ID, TOP, and HANDLER) as well as an integer TYPE field that indicates which type of structure has been returned or which type of event was generated. Programs should always check the type field before referencing fields that are not present in all property sheet event structures. The different property sheet widget event structures are described below.

Change Event (TYPE=0)

This event is generated whenever the user enters a new value for a property. It is also used to signal that a user-defined property needs to be changed. The following statement defines the event structure returned by the WIDGET_EVENT function:

```
{WIDGET_PROPSHEET_CHANGE, ID:0L, TOP:0L, HANDLER:0L, TYPE:0L,
   COMPONENT:OBJREF, IDENTIFIER:"", PROPTYPE:0L, SET_DEFINED: OL}
```

The COMPONENT field contains an object reference to the object associated with the property sheet. When multiple objects are associated with the property sheet, this field indicates which object is to change.

The IDENTIFIER field specifies the value of the property’s identifier attribute. This identifier is unique among all of the component’s properties.

The PROPTYPE field indicates the type of the property (integer, string, etc.). The integer values for these types are:

- 0 = USERDEF
- 1 = BOOLEAN
- 2 = INTEGER
- 3 = FLOAT
- 4 = STRING
- 5 = COLOR
- 6 = LINESTYLE
- 7 = SYMBOL
- 8 = THICKNESS
- 9 = ENUMLIST

The SET_DEFINED field indicates whether or not an undefined property is having its value set. In most circumstances, along with its new value, the property should have its UNDEFINED attribute set to zero. If a property is never marked as undefined, this field can be ignored.

Select Event (TYPE=1)

The select event is generated whenever the current row or column in the property sheet changes. Navigation between cells is performed by clicking on a cell. When the property sheet is realized, no cell is selected.

The following statement defines the event structure returned by the WIDGET_EVENT function:

```
{WIDGET_PROPSHEET_SELECT, ID:0L, TOP:0L, HANDLER:0L, TYPE:0L,
   COMPONENT:OBJREF, IDENTIFIER:""}
```

The COMPONENT field is an object reference to the object associated with the property sheet.

The IDENTIFIER field specifies the value of the property’s identifier attribute. This identifier is unique among all properties of the component.

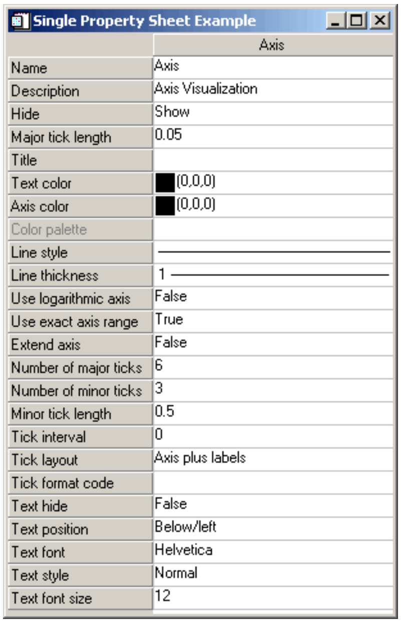
Example

Enter the following program in the IDL Editor:

```
; ExSinglePropSheet
;
; Creates a base with a property sheet. Only the
; default properties are visible. The property sheet’s
; event handler sets values and reveals selection
; changes.

PRO PropertyEvent, event

IF (event.type EQ 0) THEN BEGIN ; Value changed.

   ; Get the value of the property identified by
   ; event.identifier.
   value = WIDGET_INFO(event.id, COMPONENT = event.component, $
      PROPERTY_VALUE = event.identifier)

   ; Set the component’s property value.
   event.component -> SetPropertyByIdentifier, event.identifier, $
      value

   PRINT, 'Changed: ', event.identifier, ': ', value

ENDIF ELSE BEGIN ; Selection changed.

   PRINT, 'Selected: ' + event.identifier

ENDELSE

END

PRO ExSinglePropSheet_event, event

prop = WIDGET_INFO(event.top, $
   FIND_BY_UNAME = 'PropSheet')
WIDGET_CONTROL, prop, XSIZE = event.x, YSIZE = event.y

END

PRO CleanupEvent, baseID

WIDGET_CONTROL, baseID, GET_UVALUE = oComp
OBJ_DESTROY, oComp

END
```

```
PRO ExSinglePropSheet

; Create and initialize the component.
oComp = OBJ_NEW('IDLitVisAxis')

; Create a base and property sheet.
base = WIDGET_BASE(/TLB_SIZE_EVENT, $
   TITLE = 'Single Property Sheet Example', $
   KILL_NOTIFY = 'CleanupEvent')
prop = WIDGET_PROPERTYSHEET(base, VALUE = oComp, $
   EVENT_PRO = 'PropertyEvent', UNAME = 'PropSheet')

; Activate the widgets.
WIDGET_CONTROL, base, SET_UVALUE = oComp, /REALIZE

XMANAGER, 'ExSinglePropSheet', base, /NO_BLOCK

END
```

Save this program as `ExSinglePropSheet.pro`, then compile and run the program. A property sheet entitled Single Property Sheet Example is displayed:



Single Property Sheet Example

	Axis
Name	Axis
Description	Axis Visualization
Hide	Show
Major tick length	0.05
Title	
Text color	■ (0,0,0)
Axis color	■ (0,0,0)
Color palette	
Line style	
Line thickness	1
Use logarithmic axis	False
Use exact axis range	True
Extend axis	False
Number of major ticks	6
Number of minor ticks	3
Minor tick length	0.5
Tick interval	0
Tick layout	Axis plus labels
Tick format code	
Text hide	False
Text position	Below/left
Text font	Helvetica
Text style	Normal
Text font size	12

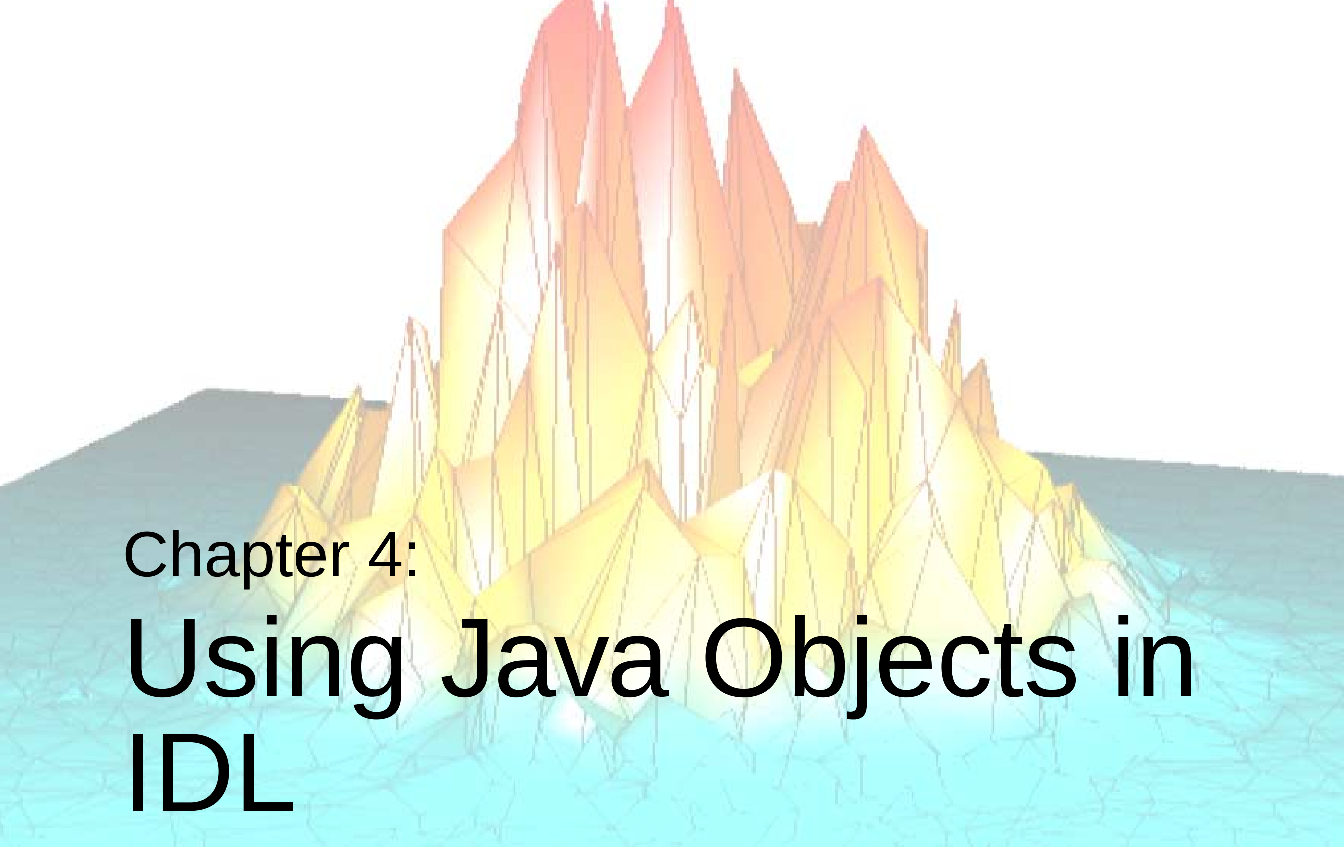
Figure 3-19: Single Property Sheet Example

For examples of the types of settings possible from the property sheet:

- Click the **Hide** setting box, click the drop-down button, and select `Hide` from the list.
- Click the **Major tick length** setting box, click the drop-down button, and move the slider to select a new value.
- Click the **Text color** setting box, click the drop-down button, and select a new color from the color selector.

Version History

Introduced: 6.0



Chapter 4: Using Java Objects in IDL

The following topics are covered in this chapter:

Overview

Java is an object-oriented programming language developed by Sun Microsystems that is commonly used for web development and other programming needs. It is beyond the scope of this chapter to describe Java in detail. Numerous third-party books and electronic resources are available. The Java website (http://java.sun.com) may be useful.

IDL 6.0 introduces the IDL-Java bridge, which allows you to access Java objects within IDL code. Java objects imported into IDL behave like normal IDL objects. See "Creating IDL-Java Objects" on page 283 for more information. The IDL-Java bridge allows the arrow operator (`->`) to be used to call the methods of these Java objects just as with other IDL objects, see "Method Calls on IDL-Java Objects" on page 285 for more information. The public data members of a Java object are accessed through GetProperty and SetProperty methods, see "Managing IDL-Java Object Properties" on page 287 for more information. These objects can also be destroyed with the OBJ_DESTROY routine, see "Destroying IDL-Java Objects" on page 289 for more information.

Note

IDL requires an evaluation or permanent IDL license to use this functionality. This functionality is not available in demo mode.

The bridge also provides IDL with access to exceptions created by the underlying Java object. This access is provided by the IDLJavaBridgeSession object, which is a Java object that maintains exceptions (errors) during a Java session, see "The IDLJavaBridgeSession Object" on page 291 for more information.

Note

Visual Java objects cannot be embedded into IDL widgets.

Currently, the IDL-Java bridge is supported on the Windows, Linux, Solaris, and Macintosh platforms supported in IDL. See "Requirements for this Release" on page 117 for more information on these platforms supported in IDL 6.0.

Java Terminology

You should become familiar with the following terms before trying to understand how IDL works with Java objects:

Java Virtual Machine (JVM) - A software execution engine for executing the byte codes in Java class files on a microprocessor.

Java Native Interface (JNI) - Standard programming interface for accessing Java native methods and embedding the JVM into native applications. For example, JNI may be used to call C/C++ functionality from Java or JNI can be used to call Java from C/C++ programs.

Java Invocation API - An API by which one may embed the Java Virtual Machine into your native application by linking the native application with the JVM shared library.

Java Reflection API - Provides a small, type-safe, and secure API that supports introspection about the classes and objects. The API can be used to:

- construct new class instances and new arrays
- access and modify fields of objects and classes
- invoke methods on objects and classes
- access and modify elements of arrays.

IDL-Java Bridge Architecture

The IDL-Java bridge uses the Java Native Interface (JNI), the reflection API, and the JVM to enable the connection between IDL and the underlying Java system.

The IDL OBJ_NEW function can be used to create a Java object. A Java-specific class token identifies the Java class used to create a Java proxy object. IDL parses this class name and creates the desired object within the underlying Java environment.

The Java-specific token is a case-insensitive form of the name of the Java class. Besides the token, the case-sensitive form of the name of the Java class is also provided because Java itself is case-sensitive while IDL is not. IDL uses the case-insensitive form to create the object definition while Java uses the case-sensitive form.

After creation, the object can then be used and manipulated just like any other IDL object. Method calls are the same as any other IDL object, but they are vectored off to an IDL Java system, which will call the appropriate Java method using JNI.

The OBJ_DESTROY procedure in IDL is used to destroy the object. This process releases the internal Java object and frees any resources associated with it.

Initializing the IDL-Java Bridge

The IDL-Java bridge must be configured before trying to create and use Java objects within IDL. The IDL program initializes the bridge when it first attempts to create an instance of IDLjavaObject. Initializing the bridge involves starting the Java Virtual Machine, creating any internal Java bridge objects (both C++ and Java) including the internal IDLJavaBridgeSession object. See "The IDLJavaBridgeSession Object" on page 291 for more information on the session object.

Configuring the Bridge

The `.idljavabrc` file on UNIX or `idljavabrc` on Windows contains the IDL-Java bridge configuration information. Even though the IDL installer attempts to create a valid working configuration file based on IDL location, the file should be verified before trying to create and use Java objects within IDL.

The IDL-Java bridge looks for the configuration file in the following order:

1. If the environment variable $IDLJAVAB_CONFIG is set, the file it indicates is used.

   Note

   This environment variable must include both the path AND the file name of the configuration file.

2. If the environment variable $IDLJAVAB_CONFIG is not set or the file indicated by that variable is not found in that location, the path specified in the $HOME environment variable is used to try to locate the configuration file.
3. If the file is not found in the path indicated by the $HOME environment variable, the <IDL_DEFAULT>`/external/objbridge/java` path is used to try to locate the configuration file.

The configuration file contains the following settings. With a text editor, open your configuration file to verify these settings are correct for your system.

- The `JVM Classpath` setting specifies additional locations for user classes. It must point to the location of any class files to be used by the bridge. On Windows, paths should be separated by semi-colons. On UNIX, colons should separate paths.

This path may contain folders that contain class files or specific jar files. It follows the same rules for specifying '-classpath' when running `java` or `javac`. You can also include the $CLASSPATH environment variable in the `JVM Classpath`:

```
JVM Classpath = $CLASSPATH:/home/johnd/myClasses.jar
```

which allows any class defined in the CLASSPATH environment variable to be used in the IDL-Java bridge.

On Windows, an example of a typical `JVM Classpath` setting is:

```
JVM Classpath = E:\myClasses.jar;$CLASSPATH
```

On UNIX, an example of a typical `JVM Classpath` setting is:

```
JVM Classpath = /home/johnd/myClasses.jar:$CLASSPATH
```

- The `JVM LibLocation` setting tells the IDL-Java bridge which JVM shared library within a given Java version to use. Various versions of Java ship with different types of JVM libraries. For example, Java 1.3 on Windows ships with a "classic" JVM, a "hotspot" JVM, and a "server" JVM. Other versions and platforms have different JVM types.

  On Windows, an example of a typical `JVM LibLocation` setting is:

  ```
  JVM LibLocation = E:\jdk1.3.1_02\jre\bin\hotspot
  ```

  On UNIX, an example of a typical `JVM LibLocation` setting is

  ```
  JVM LibLocation = /usr/java/j2re1.4.0_02/lib/sparc/client
  ```

Note

The preferred method for setting `JVM LibLocation` on Windows is via the configuration file or the IDLJAVAB_LIB_LOCATION environment variable. The preferred method on UNIX is via the $IDLJAVAB_LIB_LOCATION environment variable because UNIX requires this variable to be set in order to find Java shared libraries.

- The `JVM Option#` (where # is any whole number) setting allows you to send additional parameters to the Java Virtual machine upon initialization. These settings must be specified as string values. When these settings are encountered in the initialization, the options are added to the end of the options that the bridge sets by default.
- The `Log Location` setting indicates the directory where IDL-Java bridge log files will be created. The default location provided by the IDL installer is `/tmp` on Unix and `c:\temp` on Windows.

- The `Bridge Logging` setting indicates the type of bridge debug logging to be sent to a file called `jb_log<`*`pid`*`>.txt` (where `<`*`pid`*`>` is a process ID number) located in the directory specified by the `Log Location` setting.

  Acceptable values (from least verbose to most verbose) are `SEVERE`, `CONFIG`, `CONFIGFINE`. The default value is `SEVERE`, which specifies that bridge errors are logged. The `CONFIG` value indicates the configuration settings are also logged. The `CONFIGFINE` value is the same as `CONFIG`, but provides more detail.

  No log file is created if this setting is not specified.

The IDL-Java bridge usually only uses the configuration file once during an IDL session. The file is used when the first instance of the IDLjavaObject class is created in the session. If you edit the configuration file after the first instance is created, you must exit and restart IDL to update the IDL-Java bridge with the changes you made to the file.

IDL-Java Bridge Data Type Mapping

When data moves between IDL and a Java object, IDL automatically converts variable data types.

The following table maps how Java data types correlate to IDL data types.

Java Type (# bytes)	IDL Type	Notes
boolean (1)	Integer	True becomes 1, false becomes 0
byte (1)	Byte	
char (2)	Byte	The bridge handles Java UTF characters
short (2)	Integer	
int (4)	Long	
long (8)	Long64	
float (4)	Float	
double (8)	Double	
Java.lang.String	String	Java has the notion of a NULL string (the java.lang.String reference equals null) and the concept of an empty string. IDL makes no such differentiation, so both are identically converted.
Arrays of the above types	IDL array of the same dimensions (from 1 to 8 dimensions) and corresponding type.	

Table 4-1: Java to IDL Data Type Conversion

Java Type (# bytes)	IDL Type	Notes
Java.lang.Object (or array of java.lang.Object) and any subclass of java.lang.Object	IDL array of primitives or IDL array of IDLjavaObjects	In Java, everything is a subclass of Object. If the Java object is an array of primitives, an IDL array of the same dimensions and corresponding type (shown in this table) is created. IDL similarly converts arrays of primitives, arrays of strings, arrays of other Java objects to an IDL Java object of the same dimensions. If the Object is some single Java object, IDL creates an object reference of the IDLjavaObject class.
Null object	IDL Null object	

Table 4-1: Java to IDL Data Type Conversion (Continued)

The following table shows how data types are mapped from IDL to Java.

IDL Type	Java Type (# bytes)	Notes
Byte	byte (1)	IDL bytes range from 0 to 255, Java bytes are -128 to 127. IDL bytes converted to Java bytes will retain their binary representation but values greater than 127 will change. For example, BYTE(255) becomes a Java byte of -1. If BYTE is converted to wider Java value, the sign and value is preserved.
Integer	short (2)	
Unsigned integer	short (2)	IDL unsigned integers range from 0 to 65535, Java shorts are -32768 to 32767. IDL unsigned integers converted to Java shorts will retain their binary representation but values greater than 32768 will change. For example, UINT(65535) becomes a Java short of -1. If UINT is converted to wider Java value, the sign and value is preserved.
Long	int (4)	

Table 4-2: IDL to Java Data Type Conversion

IDL Type	Java Type (# bytes)	Notes
Unsigned long	int (4)	IDL unsigned longs range from 0 to 4294967295, Java ints are -2147483648 to 2147483647. IDL unsigned longs converted to Java ints will retain their binary representation but values greater than 2147483647 will change. For example, ULONG(4294967295) becomes a Java int of -1. If ULONG is converted to wider Java value, the sign and value is preserved.
Long64	long (8)	
Unsigned Long64	long (8)	IDL unsigned long64 range from 0 to 18446744073709551615, Java ints range from -9223372036854775808 to 9223372036854775807. IDL unsigned long64 converted to Java longs will retain their binary representation values greater than 9223372036854775807 will change. For example, ULONG64(18446744073709551615) becomes a Java long of -1.
Float	float (4)	
Double	double (8)	
String	Java.lang.String	
Arrays of the above types	Java array of the same dimensions and corresponding type	

Table 4-2: IDL to Java Data Type Conversion (Continued)

IDL Type	Java Type (# bytes)	Notes
IDLjavaObject	Object of corresponding Java class	
Arrays of objects	Java array of the same dimensions, consisting of corresponding Java proxy objects	Only objects of type IDLjavaObject are converted.
Null object	Java null	

Table 4-2: IDL to Java Data Type Conversion (Continued)

When calling a Java method or constructor from IDL, the data parameters are promoted as little as possible based on the signature of the given method. The following table shows how data types are promoted within Java relative to IDL.

Note

When strings and arrays are passed between IDL and Java, the array must be copied. Depending upon the size of the array, this copy may be time intensive. Care should be taken to minimize array copying.

IDL Type	Java Type (to order of desired promotion)	Notes
Byte	byte, char, short, int, long, float, double, boolean	
Integer	short, int, long, float, double, boolean	
Unsigned integer	short, int, long, float, double, boolean	
Long	int, long, float, double, boolean	
Unsigned Long	int, long, float, double, boolean	
Long64	long, float, double, boolean	
Unsigned Long64	long, float, double, boolean	

Table 4-3: Java Data Type Promotion Relative to IDL

IDL Type	Java Type (to order of desired promotion)	Notes
Float	float, double	
Double	double	
String	Java.lang.String	
IDLjavaObject	Java.lang.Object	

Table 4-3: Java Data Type Promotion Relative to IDL (Continued)

Creating IDL-Java Objects

As with all IDL objects, a Java object is created using the IDL OBJ_NEW function. Keying off the provided Java class name, the underlying implementation uses the IDL Java subsystem to call the constructor on the desired Java object. The following line of code demonstrates the basic syntax for calling OBJ_NEW to create a Java object within IDL:

```
oJava = OBJ_NEW(IDLjavaObject$JAVACLASSNAME, JavaClassName, $
   [Arg1, Arg2, ..., ArgN])
```

where *JAVACLASSNAME* is the class name token used by IDL to create the object, *JavaClassName* is the class name used by Java to initialize the object, and *Arg1* through *ArgN* are any data parameters required by the constructor. See "Java Class Names in IDL" for more information.

See the `hellojava.pro` file in the `external/objbridge/java/examples` directory of the IDL distribution for a simple example of an IDL-Java object creation.

Note

If you edit and recompile a Java class used by IDL during an IDL-Java bridge session, you must first exit and restart IDL before your modified Java class will be recognized by IDL.

The IDL-Java bridge also provides the ability to access static Java methods and data members. See "Java Static Access" on page 284 for more information.

Java Class Names in IDL

The underlying Java interpreter recognizes the Java class name including all objects contained within the Java interpreter's class path.

To identify a proper Java object, the fully-qualified package name should be used when creating the IDL class name. For example, a class of type String would be referred to as `java.lang.String`.

In the IDL class name, the Java class separator ('.') should be replaced with an underscore ('_'). If a Java class of type String were created, the following IDL OBJ_NEW call would be used:

```
oJString = OBJ_NEW('IDLJavaObject$JAVA_LANG_STRING',$
   'java.lang.String', 'My String')
```

The class name is provided twice because IDL is case-insensitive whereas Java is case-sensitive, see "IDL-Java Bridge Architecture" on page 273 for more information.

Note

IDL objects use method names (INIT and CLEANUP) to identify and call object lifecycle methods. As such, these method names should be considered reserved. If an underlying Java object implements a method using either INIT or CLEANUP, those methods will be overridden by the IDL methods and not accessible from IDL. In Java, you can wrap these methods with different named methods to work around this limitation.

Java Static Access

In Java, a program can call a static method or access static data members on a Java class without first having to create the object.

IDL contains a special wrapper object type for calling static methods. This IDL object wrapper references the underlying Java class, allowing the object to call static methods on the class or allowing the object to use the Get/Set Property calls to access static data members. The following line of code demonstrates the basic syntax for calling OBJ_NEW to create a static proxy within IDL:

```
oJava = OBJ_NEW(IDLjavaObject$Static$JAVACLASSNAME, JavaClassName)
```

where *JAVACLASNAME* is the class name token used by IDL to create the object and *JavaClassName* is the class name used by Java to initialize the object. See "Java Class Names in IDL" on page 283 for more information.

A special static object would not need to be created to call an instantiated `IDLJavaObject` with static methods:

```
oNotStatic = OBJ_NEW('IDLjavaObject$JAVACLASSNAME', $
   'JavaClassName')
oNotStatic -> aStaticMethod ; this is OK
```

See the `javaprops.pro` file in the `external/objbridge/java/examples` directory of the IDL distribution for an example of working with static data members.

Note

All restrictions on creating Java objects apply to this static object.

Method Calls on IDL-Java Objects

When a method is called on a Java-based IDL object, the method name and arguments are passed to the IDL-Java subsystem and the Java Reflection API to construct and invoke the method call on the underlying object.

IDL handles conversion between IDL and Java data types. Any results are returned in IDL variables of the appropriate type.

As with all IDL objects, the general syntax in IDL for an underlying Java method that returns a value (known as a function method in IDL) is:

```
result = ObjRef -> Method([Arguments])
```

and the general syntax in IDL for an underlying Java method that does not return a value, a void method, (known as a procedure method in IDL) is:

```
ObjRef -> Method[, Arguments]
```

where `ObjRef` is an object reference to an instance of a dynamic subclass of the IDLjavaObject class.

Note

Besides other Java based objects, the value of an argument may be an IDL primitive type, an IDLjavaObject, or an IDL primitive type array. No complex types (structures, pointers, etc.) are supported as parameters to method calls.

What Happens When a Method Call is Made?

When a method is called on an instance of IDLjavaObject, IDL uses the method name and arguments to construct the appropriate method calls for the underlying Java object.

From the point of view of an IDL user issuing method calls on an instance of IDLjavaObject, this process is completely transparent. IDL handles the translation when the IDL user calls the Java object’s method.

Due to case-sensitivity incompatibilities between IDL and Java, Java’s ability to overload methods, and the fact that Java might promote certain data types, the Java bridge uses an algorithm to match the IDL method name and parameters to the corresponding Java object method.

Before the algorithm starts, IDL provides a case-insensitive <METHODNAME> and a reference to the Java object. For a given object and its parent classes, the Java bridge obtains a list of all the public method names, including static methods. This algorithm performs the following steps:

1. If the Java class has one method name matching the IDL <METHODNAME> (except for case insensitivity), this Java method name is used. At this point, signatures and overloaded functions are not taken into account.
2. If the Java class has several method names that differ only in case and one is all uppercase, the uppercase name is used. Otherwise, the IDL-Java bridge issues an error that it has no method named <METHODNAME>.
3. Once the method name has been determined, a promotion algorithm then matches the Java data parameters as closely as possible with the IDL parameters. Minimum data promotion from IDL to Java is preferred and only widening promotion is allowed. If no match is found, an error is issued.

Data Type Conversions

IDL and Java use different data types. IDL's dynamic type conversion facilities handle all conversion of data types between IDL and the Java system. The data type mappings are described in "IDL-Java Bridge Data Type Mapping" on page 277.

For example, if the Java object has a method that requires a value of type `int` as an input argument, IDL would supply the value as an IDL Long. For any other IDL data type, IDL would first convert the value to an IDL Long using its normal data type conversion mechanism before passing the value to the Java object as an `int`.

Managing IDL-Java Object Properties

Property names and arguments are also passed to the IDL Java subsystem and are used in conjunction with the Java Reflection API to construct and access public data members on the underlying object. These public data members (known as properties in IDL) are identified through arguments to the GetProperty and SetProperty methods. See "Getting and Setting Properties" on page 288 for more information.

Note

Only public data members may be accessed.

Due to case-sensitivity incompatibilities between IDL and Java and the fact that Java might promote certain data types, the Java bridge uses an algorithm to match the IDL properties name to the corresponding Java object data members.

Before the algorithm starts, IDL provides a case-insensitive <PROPERTYNAME> and a reference to the Java object. For the given object and its parent classes, the Java bridge obtains a list of all the public data members including static members. This algorithm performs the following steps:

1. If the Java class has one data member name matching the IDL <PROPERTYNAME> (except for case insensitivity), this Java data member is used. At this point, data types are not yet taken into account; this algorithm only matches the data member names.
2. If the Java class has several member names that differ only in case, the data member name that exactly matches the IDL < PROPERTYNAME > (i.e. the one that is all caps) is called. Otherwise, the IDL-Java bridge issues an error that the class has no data members named < PROPERTYNAME >.
3. When setting a property with the SetProperty method, a promotion algorithm matches the provided IDL parameter with the Java data parameter as closely as possible. If the IDL value can be promoted to the same type as the data member, this data member is used. Otherwise, an error is issued.

   When retrieving a property with the GetProperty method, this step is skipped and the value is returned to IDL.

See the `allprops.pro` and `publicmembers.pro` files in the `external/objbridge/java/examples` directory of the IDL distribution for IDL routines that provide information about data members associated with given Java classes.

Getting and Setting Properties

The IDL-Java bridge follows the standard IDL property interface to support data member access on Java objects and classes.

To retrieve a property value from a Java object, use the following syntax:

```
ObjRef -> GetProperty, PROPERTY=variable
```

where `ObjRef` is an instance of IDLjavaObject that encapsulates the Java object, *PROPERTY* is the name of the Java object's data member (property), and *variable* is the name of an IDL variable that will contain the retrieved property value.

To retrieve multiple property values in a single statement supply multiple *PROPERTY*=*variable* pairs separated by commas.

To set a property value on a Java object, use the following syntax:

```
ObjRef -> SetProperty, Property=value
```

where `ObjRef` is an instance of IDLjavaObject that encapsulates the Java object, *PROPERTY* is the name of the Java object's data member, and *value* is value of the property to be set.

To set multiple property values in a single statement supply multiple *PROPERTY*=*value* pairs separated by commas.

Note

The provided *PROPERTY* must map directly to a data member name. Any name passed into either of the property routines is assumed to be a fully qualified Java property name. As such, the partial property name functionality provided by IDL is not valid with IDL Java based objects.

The *variable* or *value* part may be an IDL primitive type, an instance of IDLJavaObject, or an array of an IDL primitive type. See "IDL-Java Bridge Data Type Mapping" on page 277 for more information.

Note

Besides other Java based objects, no complex types (Structures, pointers, etc.) are supported as parameters to property calls.

Destroying IDL-Java Objects

The OBJ_DESTROY routine is used to destroy instances of IDLjavaObject. When OBJ_DESTROY is called with a Java based object as an argument, IDL releases the underlying Java object and frees IDL resources relating to that object.

Note

Destruction of the IDL object does not automatically cause the destruction of the underlying Java object. Because Java utilizes a garbage collection mechanism to release any information allocated for a particular object, the resources utilized by the underlying Java object will persist until the Java virtual machine’s garbage collector runs.

Showing IDL-Java Output in IDL

By default, IDL prints the output from Java (the `System.out` and `System.err` output streams).

For example, given the following Java code:

```
public class helloWorld
{
 // ctor
 public helloWorld() {
  System.out.println("helloWorld ctor");
   }

 public void sayHello() {
  System.out.println("Hello! (from the helloWorld object)");
   }

}
```

The following output occurs in IDL:

```
IDL> oJHello = OBJ_NEW('IDLjavaObject$HelloWorld', 'helloWorld')
% helloWorld ctor
IDL> oJHello -> SayHello
% Hello! (from the helloWorld object)
IDL> OBJ_DESTROY, oJHello
```

This example code is also provided in the `helloJava.java` and `hellojava2.pro` files, which are in the `external/objbridge/java/examples` directory of the IDL distribution.

Note

Due to restrictions in IDL concerning receiving standard output from non-main threads, the bridge will only send `System.out` and `System.err` information to IDL from the main thread. Other thread’s output will be ignored.

Note

A `print()` in Java will not have a carriage return at the end of the line (as opposed to `println()`, which does). However, when outputting to Java both `print()` and `println()` will print to IDL followed by a carriage return. You can change this result by having the Java-side application buffer its data up into the lines you wish to see on the IDL-side.

The IDLJavaBridgeSession Object

Java exceptions are handled within IDL through an IDL-Java bridge session object, IDLJavaBridgeSession. This Java object can be queried to determine the status of the bridge, including information on any exceptions. For example, one important Java object available through the session object is the last issued Java exception.

The session object is a proxy to an internal Java object, which is created during the IDL-Java bridge initialization process. You can connect an IDLJavaObject to this object using OBJ_NEW:

```
oJSession = OBJ_NEW('IDLjavaObject$IDLJAVABRIDGESESSION’)
```

Note

Only one Java session object needs to be created during an IDL session. Subsequent calls to this object will point to the same internal object.

When an exception occurs, the GetException function method indicates what exception occurred:

```
oJException = oJSession -> GetException()
```

where `oJSession` is a reference to the session object and `oJException` is a proxy object to a `java.lang.Throwable` object, which is the class used in Java to manage exceptions. The session object also has a ClearException method that clears the session object’s last exception. The GetException method always calls ClearException method.

The IDLJavaBridgeSession object also has the GetVersionObject method, which retrieves the IDLJavaVersion object:

```
oJVersion = oJSession -> GetVersionObject()
```

where `oJSession` is a reference to the session object and `oJVersion` is a proxy object to an IDLJavaVersion object. This object determines version information about the IDL-Java bridge and the underlying Java system.

The IDLJavaVersion object provides the following function methods, which do not require any arguments:

- GetBuildDate() - a java.lang.String object specifying the build date. For example, `Apr 1 2003`.
- GetJavaVersion() - a java.lang.String object specifying the Java version. For example, `1.3.1_02`.
- GetBridgeVersion() - a java.lang.String object specifying the IDL-Java bridge version.

An example of the version object is provided in the `bridge_version.pro` file, which is in IDL's `external/objbridge/java/examples` directory.

Java Exceptions

During the operation of the bridge, an error may occur when initializing the bridge, creating an `IDLjavaObject`, calling methods, setting properties, or getting properties. Typically, these errors will be fixed by changing your IDL or Java code (or by changing the bridge configuration). Java bridge errors operate like other IDL errors in that they stop execution of IDL and post an error message. These errors can be caught like any other IDL error.

On the other hand, Java uses the exception mechanism to report errors. For example, in Java, if we attempt to create a java.lang.StringBuffer of negative length, a java.lang.NegativeArraySizeException is issued.

Java exceptions are handled much like bridge errors. They stop IDL execution (if uncaught) and they report an error message containing a line number. In addition, a mechanism is provided to grab the exception object (a subclass of java.lang.Throwable) via the session object. Once connected with the exception object, IDL can call any of the methods provided by this Java object. For example, IDL can query the exception name to determine how to handle it, or print a stack trace of where the exception occurred in your Java code.

The exception object is provided through the GetExpection method to the IDLJavaBridgeSession object. See "The IDLJavaBridgeSession Object" on page 291 for more information about this object.

Uncaught Exceptions

If a Java exception is not caught, IDL will stop execution and display an `Exception thrown` error message. For example, when the following program is saved as `ExceptIssued.pro`, compiled, and ran in IDL:

```
PRO ExceptIssued

; This will throw a Java exception
oJStrBuffer = OBJ_NEW($
   'IDLJavaObject$java_lang_StringBuffer', $
   ’java.lang.StringBuffer’, -2)

END
```

IDL issues the following output:

```
IDL> ExceptIssued
% Exception thrown
% Execution halted at: EXCEPTISSUED 4 ExceptIssues.pro
%                      $MAIN$
```

From the IDL command line, you can then use the session object to help debug the problem:

```
IDL> oJSession = OBJ_NEW('IDLJavaObject$IDLJAVABRIDGESESSION')
IDL> oJExc = oJSession -> GetException()
IDL> oJExc -> PrintStackTrace
% java.lang.NegativeArraySizeException:
%    at java.lang.StringBuffer.<init>(StringBuffer.java:116)
```

A similar example is also provided in the `exception.pro` file, which is in the `external/objbridge/java/examples` directory of the IDL distribution. The `exception.pro` example shows how to use the utility routine provided in the `showexcept.pro` file. This `showexcept` utility routine can be re-used to provide consist error messages when Java exceptions occur. The `showexcept.pro` file is also provided in the `external/objbridge/java/examples` directory of the IDL distribution.

Caught Exceptions

Java exceptions can be caught just like IDL errors. Consult the documentation of the Java classes that you are using to ensure IDL is catching any expected exceptions. For example:

```
PRO ExceptCaught

; Grab the special IDLJavaBridgeSession object
oJBridgeSession = OBJ_NEW('IDLJavaObject$IDLJAVABRIDGESESSION')

bufferSize = -2
; Our Java constructor might throw an exception, so let’s catch it
CATCH, error_status
IF (error_status NE 0) THEN BEGIN
   ; Use session object to get our Exception
   oJExc = oJBridgeSession -> GetException()
   ; should be of type
   ; IDLJAVAOBJECT$JAVA_LANG_NEGATIVEARRAYSIZEEXCEPTION
   HELP, oJExc
   ; Now we can access the members java.lang.Throwable
   PRINT, 'Exception thrown:', oJExc -> ToString()
   oJExc -> PrintStackTrace
   ; Cleanup
   OBJ_DESTROY, oJExc
```

```
      ; Increase the buffer size to avoid the exception.
      bufferSize = bufferSize + 100
   ENDIF

   ; This throws a Java exception the 1st time, but pass the 2nd time.
   oJStrBuffer = OBJ_NEW('IDLJavaObject$java_lang_StringBuffer', $
      'java.lang.StringBuffer', bufferSize)


   OBJ_DESTROY, oJStrBuffer
   OBJ_DESTROY, oJBridgeSession

   END
```

A similar example is also provided in the `exception.pro` file, which is in the `external/objbridge/java/examples` directory of the IDL distribution. The `exception.pro` example shows how to use the utility routine provided in the `showexcept.pro` file. This `showexcept` utility routine can be re-used to provide consist error messages when Java exceptions occur. The `showexcept.pro` file is also provided in the `external/objbridge/java/examples` directory of the IDL distribution.

IDL-Java Bridge Examples

The following examples demonstrate how to access data through the IDL-Java bridge:

- "Accessing Arrays Example"
- "Accessing URLs Example" on page 299
- "Accessing Grayscale Images Example" on page 301
- "Accessing RGB Images Example" on page 304

Note

If IDL is not able to find any Java class associated with these examples, make sure your IDL-Java bridge is properly configured. See "Configuring the Bridge" on page 274 for more information.

Accessing Arrays Example

This example creates a two-dimensional array within a Java class, which is contained in a file named `array2d.java`. IDL then accesses this data through the ArrayDemo routine, which is in a file named `arraydemo.pro`. These files are also in the IDL distribution within the `external/objbridge/java/examples` directory.

The `array2d.java` file contains the following text for creating a two-dimensional array in Java:

```
public class array2d
{
 short[][]   m_as;
 long[][]    m_aj;

 // ctor
 public array2d() {
   int SIZE1 = 3;
   int SIZE2 = 4;

   // default ctor creates a fixed number of elements
   m_as = new short[SIZE1][SIZE2];
   m_aj = new long[SIZE1][SIZE2];

   for (int i=0; i<SIZE1; i++) {
     for (int j=0; j<SIZE2; j++) {
       m_as[i][j] = (short)(i*10+j);
       m_aj[i][j] = (long)(i*10+j);
```

```
        }
      }

    }


    public void setShorts(short[][] _as) {
      m_as = _as;
    }
    public short[][] getShorts() {return m_as;}
    public short getShortByIndex(int i, int j) {return m_as[i][j];}


    public void setLongs(long[][] _aj) {
      m_aj = _aj;
    }
    public long[][] getLongs() {return m_aj;}
    public long getLongByIndex(int i, int j) {return m_aj[i][j];}


   }
```

The `arraydemo.pro` file contains the following text for accessing the two-dimensional array within IDL:

```
PRO ArrayDemo

; The Java class array2d creates 2 initial arrays, one
; of longs and one of shorts. We can interrogate and
; change this array.
oJArr = OBJ_NEW('IDLJavaObject$ARRAY2D', 'array2d')

; First, let’s see what is in the short array at index
; (2,3).
PRINT, 'array2d short(2, 3) = ', $
   oJArr -> GetShortByIndex(2, 3), $
   '    (should be 23)’

; Now, let’s copy the entire array from Java to IDL.
shortArrIDL = oJArr -> GetShorts()
HELP, shortArrIDL
PRINT, 'shortArrIDL[2, 3] = ', shortArrIDL[2, 3], $
   '    (should be 23)'

; Let’s change this value...
shortArrIDL[2, 3] = 999
; ...and copy it back to Java...
oJArr -> SetShorts, shortArrIDL
; ...now its value should be different.
```

```
PRINT, 'array2d short(2, 3) = ', $
   oJArr -> GetShortByIndex(2, 3), '    (should be 999)'

; Let's set our array to something different.
oJArr -> SetShorts, INDGEN(10, 8)
PRINT, 'array2d short(0, 0) = ', $
   oJArr -> GetShortByIndex(0, 0), '    (should be 0)'
PRINT, 'array2d short(1, 0) = ', $
   oJArr -> GetShortByIndex(1, 0), '    (should be 1)'
PRINT, 'array2d short(2, 0) = ', $
   oJArr -> GetShortByIndex(2, 0), '    (should be 2)'
PRINT, 'array2d short(0, 1) = ', $
   oJArr -> GetShortByIndex(0, 1), '    (should be 10)'

; Array2d has a setLongs method, but b/c arrays do not
; (currently) promote, the first call to setLongs works
; but the second fails.
oJArr -> SetLongs, L64INDGEN(10, 8)
PRINT, 'array2d long(0, 1) = ', $
   oJArr -> GetLongByIndex(0, 1), '    (should be 10)'

;PRINT, '(expecting an error on the next line...)'
;oJArr -> SetLongs, INDGEN(10,8)

; Cleanup our object.
OBJ_DESTROY, oJArr

END
```

After saving and compiling the above files (`array2d.java` in Java and `ArrayDemo.pro` in IDL), update the `jbexamples.jar` file in the `external/objbridge/java` directory with the new compiled class and run the ArrayDemo routine in IDL. The routine should produce the following results:

```
array2d short(2, 3) = 23 (should be 23)
SHORTARRIDL   INT = Array[3, 4]
shortArrIDL[2, 3] = 23 (should be 23)
array2d short(2, 3) = 999 (should be 999)
array2d short(0, 0) = 0 (should be 0)
array2d short(1, 0) = 1 (should be 1)
array2d short(2, 0) = 2 (should be 2)
array2d short(0, 1) = 10 (should be 10)
array2d long(0, 1) = 10 (should be 10)
```

Accessing URLs Example

This example finds and reads a given URL, which is contained in a file named `URLReader.java`. IDL then accesses this data through the URLRead routine, which is in a file named `urlread.pro`. These files are also in the IDL distribution within the `external/objbridge/java/examples` directory.

The `URLReader.java` file contains the following text for reading a given URL in Java:

```
import java.io.*;
import java.net.*;

public class URLReader
{
  private ByteArrayOutputStream m_buffer;

  // ***********************************************************
  //
  // Constructor.  Create the reader
  //
  // ***********************************************************
   public URLReader() {
      m_buffer = new ByteArrayOutputStream();
   }

  // ***********************************************************
  //
  // readURL: read the data from the URL into our buffer
  //
  //    returns: number of bytes read (0 if invalid URL)
  //
  // NOTE: reading a new URL clears out the previous data
  //
  // ***********************************************************
   public int readURL(String sURL) {
      URL url;
      InputStream in = null;


      m_buffer.reset();  // reset our holding buffer to 0 bytes

      int total_bytes = 0;
      byte[] tempBuffer = new byte[4096];
      try {
         url = new URL(sURL);
         in = url.openStream();
         int bytes_read;
         while ((bytes_read = in.read(tempBuffer)) != -1) {
            m_buffer.write(tempBuffer, 0, bytes_read);
            total_bytes += bytes_read;
         }
      } catch (Exception e) {
         System.err.println("Error reading URL: "+sURL);
         total_bytes = 0;
      } finally {
         try {
            in.close();
            m_buffer.close();
         } catch (Exception e) {}
      }

      return total_bytes;
   }

  // ********************************************************
  //
  // getData: return the array of bytes
  //
  // ********************************************************
   public byte[] getData() {
     return m_buffer.toByteArray();
   }

  // ********************************************************
  //
  // main: reads URL and reports # of byts reads
  //
  //   Usage: java URLReader <URL>
  //
  // ********************************************************

  public static void main(String[] args) {
     if (args.length != 1)
        System.err.println("Usage: URLReader <URL>");
     else {
        URLReader o = new URLReader();
        int b = o.readURL(args[0]);
        System.out.println("bytes="+b);
     }
  }



}
```

The `urlread.pro` file contains the following text for inputting an URL as an IDL string and then accessing its data within IDL:

```
FUNCTION URLRead, sURLName

; Create an URLReader.
oJURLReader = OBJ_NEW('IDLjavaObject$URLReader', 'URLReader')

; Read the URL data into our Java-side buffer.
nBytes = oJURLReader -> ReadURL(sURLName)

;PRINT, 'Read ', nBytes, ' bytes'

; Pull the data into IDL.
byteArr = oJURLReader -> GetData()

; Cleanup Java object.
OBJ_DESTROY, oJURLReader

; Return the data.
RETURN, byteArr

END
```

After saving and compiling the above files (`URLReader.java` in Java and `urlread.pro` in IDL), you can run the URLRead routine in IDL. This routine is a function with one input argument, which should be a IDL string containing an URL. For example:

```
address = 'http://www.RSInc.com'
data = URLRead(address)
```

Accessing Grayscale Images Example

This example creates a a grayscale ramp image within a Java class, which is contained in a file named `GreyBandsImage.java`. IDL then accesses this data through the ShowGreyImage routine, which is in the `showgreyimage.pro` file. These files are also in the IDL distribution within the `external/objbridge/java/examples` directory.

The `GreyBandsImage.java` file contains the following text for creating a grayscale image in Java:

```
import java.awt.*;
import java.awt.image.*;

public class GreyBandsImage extends BufferedImage
{
 // Members
 private int m_height;
 private int m_width;


 //
 // ctor
 //
 public GreyBandsImage() {
    super(100, 100, BufferedImage.TYPE_INT_ARGB);
    generateImage();
    m_height = 100;
    m_width = 100;
 }

 //
 // private method to generate the image
 //
 private void generateImage() {
    Color c;
    int width  = getWidth();
    int height = getHeight();
    WritableRaster raster = getRaster();
    ColorModel model = getColorModel();

    int BAND_PIXEL_WIDTH = 5;
    int nBands = width/BAND_PIXEL_WIDTH;
    int greyDelta = 255 / nBands;
    for (int i=0 ; i < nBands; i++) {
          c = new Color(i*greyDelta, i*greyDelta, i*greyDelta);
          int argb = c.getRGB();
          Object colorData = model.getDataElements(argb, null);

          for (int j=0; j < height; j++)
             for (int k=0; k < BAND_PIXEL_WIDTH; k++)
                raster.setDataElements(j, (i*5)+k, colorData);

    }
 }
 //
 // mutators
 //
 public int[] getRawData() {
   Raster oRaster = getRaster();
   Rectangle oBounds = oRaster.getBounds();
   int[] data = new int[m_height * m_width * 4];

   data = oRaster.getPixels(0,0,100,100, data);
   return data;
 }
 public int getH() {return m_height; }
 public int getW() {return m_width; }


}
```

The `showgreyimage.pro` file contains the following text for accessing the grayscale image within IDL:

```
PRO ShowGreyImage

; Construct the GreyBandImage in Java. This is a sub-class of
; BufferedImage. It is actually a 4 band image that happens to
display bands in greyscale. It is 100x100 pixels.
oGrey = OBJ_NEW('IDLjavaObject$GreyBandsImage', 'GreyBandsImage')

; Get the 4 byte pixel values.
data = oGrey -> GetRawData()

; Get the height and width.
h = oGrey -> GetH()
w = oGrey -> GetW()

; Display the graphic in an IDL window
WINDOW, 0, XSIZE = 100, YSIZE = 100
TV, REBIN(data, h, w)

; Cleanup
OBJ_DESTROY, oGrey

END
```

After saving and compiling the above files (`GreyBandsImage.java` in Java and `showgreyimage.pro` in IDL), you can run the ShowGreyImage routine in IDL. The routine should produce the following image:

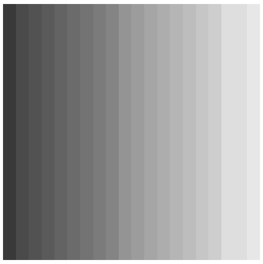


Figure 4-1: Java Grayscale Image Example

Accessing RGB Images Example

This example imports an RGB (red, green, and blue) image from the IDL distribution into a Java class. The image is in the `glowing_gas.jpg` file, which is in the `examples/data` directory of the IDL distribution. The Java class also displays the image in a Java Swing user-interface. Then, the image is accessed into IDL and displayed with the new iImage tool. The Java and IDL code for this example is provided in the `external/objbridge/java/examples` directory, but the Java code has not been built as part of the `jbexamples.jar` file.

Note

This example uses functionality only available in Java 1.4 and later.

Note

Due to a Java bug, this example (and any other example using Swing on AWT) will not work on Linux platforms.

The first and main Java class is FrameTest, which creates the Java Swing application that imports the image from the `glowing_gas.jpg` file. Copy and paste the following text into a file, then save it as `FrameTest.java`:

```
import java.awt.*;
import java.awt.image.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import java.io.File;

public class FrameTest extends JFrame {

 RSIImageArea c_imgArea;
 int m_xsize;
 int m_ysize;
 Box c_controlBox;

 public FrameTest() {

  super("This is a JAVA Swing Program called from IDL");
  // Dispose the frame when the sys close is hit
  setDefaultCloseOperation(DISPOSE_ON_CLOSE);
  m_xsize = 350;
  m_ysize = 371;
  buildGUI();

 }

 public void buildGUI() {

  c_controlBox = Box.createVerticalBox();

  JLabel l1 = new JLabel("Example Java/IDL Interaction");
  JButton bLoadFile = new JButton("Load new file");
  bLoadFile.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
    JFileChooser chooser = new JFileChooser(new
     File("c:\\RSI\\IDL60\\EXAMPLES\\DATA"));
    chooser.setDialogTitle("Enter a JPEG file");
    if (chooser.showOpenDialog(FrameTest.this) ==
     JFileChooser.APPROVE_OPTION) {

      java.io.File fname = chooser.getSelectedFile();
      String filename = fname.getPath();
      System.out.println(filename);
      c_imgArea.setImageFile(filename);
    }
   }
  });

  JButton b1 = new JButton("Close this example");
  b1.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
    dispose();
   }
  });

  c_imgArea = new
   RSIImageArea("c:\\rsi\\idl60\\examples\\data\\glowing_gas.jpg",
    new Dimension(m_xsize,m_ysize));



  Box mainBox = Box.createVerticalBox();
  Box rowBox = Box.createHorizontalBox();
  rowBox.add(b1);
  rowBox.add(bLoadFile);

  c_controlBox.add(l1);
  c_controlBox.add(rowBox);
  mainBox.add(c_controlBox);
  mainBox.add(c_imgArea);

  getContentPane().add(mainBox);

  pack();
  setVisible(true);
  c_imgArea.displayImage();
  c_imgArea.addResizeListener(new RSIImageAreaResizeListener() {
   public void areaResized(int newx, int newy) {
    Dimension cdim = c_controlBox.getSize(null);
    Insets i = getInsets();
    newx = i.left + i.right + newx;
    newy = i.top + cdim.height + newy + i.bottom;
    setSize(new Dimension(newx, newy));
   }
  });
 }

 public void setImageData(int [] imgData, int xsize, int ysize) {
  MemoryImageSource ims = new MemoryImageSource(xsize, ysize,
   imgData, 0, ysize);
  Image imgtmp = createImage(ims);
  Graphics g = c_imgArea.getGraphics();
  g.drawImage(imgtmp, 0, 0, null);

 }
```

```
public void setImageData(byte [][][] imgData, int xsize,
 int ysize) {


 System.out.println("SIZE = "+xsize+"x"+ysize);
 int newArray [] = new int[xsize*ysize];
 int pixi = 0;
 int curpix = 0;
 short [] currgb = new short[3];
 for (int i=0;i<m_xsize;i++) {
  for (int j=0;j<m_ysize;j++) {
   for (int k=0;k<3;k++) {
    currgb[k] = (short) imgData[k][i][j];
    currgb[k] = (currgb[k] < 128) ? (short) currgb[k] : (short)
     (currgb[k]-256);
   }
   curpix = (int) currgb[0] *  +
    ((int) currgb[1] * (int) Math.pow(2,8)) +
     ((int) currgb[2] * (int) Math.pow(2,16));
   if (pixi % 1000 == 0)
    System.out.println("PIXI = "+pixi+" "+curpix);
   newArray[pixi++] = curpix;
  }
 }

 MemoryImageSource ims = new MemoryImageSource(xsize, ysize,
  newArray, 0, ysize);
 c_imgArea.setImageObj(c_imgArea.createImage(ims));

}

public byte[][][] getImageData()
{
 int width = 1;
 int height = 1;
 PixelGrabber pGrab;

 width = m_xsize;
 height = m_ysize;

 // pixarray for the grab - 3D bytearray for display
 int [] pixarray = new int[width*height];
 byte [][][] bytearray = new byte[3][width][height];


 // create a pixel grabber
 pGrab = new PixelGrabber(c_imgArea.getImageObj(),0,0,
      width,height, pixarray, 0, width);

      // grab the pixels from the image
      try {
       boolean b = pGrab.grabPixels();
      } catch (InterruptedException e) {
       System.err.println("pixel grab interrupted");
       return bytearray;
      }

      // break down the 32-bit integers from the grab into 8-bit bytes
      // and fill the return 3D array
      int pixi = 0;
      int curpix = 0;
      for (int j=0;j<m_ysize;j++) {
       for (int i=0;i<m_xsize;i++) {
        curpix = pixarray[pixi++];
        bytearray[0][i][j] = (byte) ((curpix >> 16) & 0xff);
        bytearray[1][i][j] = (byte) ((curpix >>  8) & 0xff);
        bytearray[2][i][j]  = (byte) ((curpix      ) & 0xff);
       }
      }
      return bytearray;
     }


     public static void main(String [] args) {
      FrameTest f = new FrameTest();
     }

    }
```

Note

The above text is for the FrameTest class that accesses the `glowing_gas.jpg` file in the `examples/data` directory of a default installation of IDL on a Windows system. The file's location is specified as `c:\\RSI\\IDL60\\EXAMPLES\\DATA` in the above text. If the `glowing_gas.jpg` file is not in the same location on system, edit the text to change the location of this file to match your system.

The FrameTest class uses two other user-defined classes, RSIImageArea and RSIImageAreaResizeListener. These classes help to define the viewing area and display the image in Java. Copy and paste the following text into a file, then save it as `RSIImageArea.java`:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.Vector;
import java.io.File;

public class RSIImageArea extends JComponent implements
 MouseMotionListener, MouseListener {


 Image c_img;
 int m_boxw = 100;
 int m_boxh = 100;
 Dimension c_dim;
 boolean m_pressed = false;
 int m_button = 0;
 Vector c_resizelisteners = null;

 public RSIImageArea(String imgFile, Dimension dim) {

  c_img = getToolkit().getImage(imgFile);
  c_dim = dim;
  setPreferredSize(dim);
  setSize(dim);
  addMouseMotionListener(this);
  addMouseListener(this);

 }

 public void addResizeListener(RSIImageAreaResizeListener l) {
  if (c_resizelisteners == null) c_resizelisteners = new Vector();
  if (! c_resizelisteners.contains(l))  c_resizelisteners.add(l);
 }
 public void removeResizeListener(RSIImageAreaResizeListener l) {
  if (c_resizelisteners == null) return;
  if (c_resizelisteners.contains(l)) c_resizelisteners.remove(l);
 }

 public void displayImage() {
  repaint();
 }

 public void paint(Graphics g) {
```

```
 int xsize = c_img.getWidth(null);
 int ysize = c_img.getHeight(null);
 if (xsize != -1 && ysize != -1) {
  if (xsize != c_dim.width || ysize != c_dim.height) {
   c_dim.width = xsize;
   c_dim.height = ysize;
   setPreferredSize(c_dim);
   setSize(c_dim);
   if (c_resizelisteners != null) {
    RSIImageAreaResizeListener l = null;
    for (int j=0;j<c_resizelisteners.size();j++) {
     l = (RSIImageAreaResizeListener)
      c_resizelisteners.elementAt(j);
     l.areaResized(xsize, ysize);
    }
   }
  }
 }
 g.drawImage(c_img, 0, 0, null);
}

public void setImageFile(String fileName) {
 c_img = null;
 c_img = getToolkit().getImage(fileName);
 repaint();
}


public Image getImageObj() {
 return c_img;
}

public void setImageObj(Image img) {
 c_img = img;
 repaint();
}

public void drawZoomBox(MouseEvent e) {
 int bx = e.getX() - m_boxw/2;
 bx = (bx >=0) ? bx :0;
 int by = e.getY() - m_boxh/2;
 by = (by >=0) ? by :0;
 int ex = bx + m_boxw;
 if (ex > c_dim.width) {
  ex = c_dim.width;
  bx = c_dim.width-m_boxw;
 }
 int ey = by + m_boxh;
 if (ey > c_dim.height) {
  ey = c_dim.height;
  by = c_dim.height-m_boxh;
 }

 repaint();
 Graphics g = getGraphics();
 g.drawImage(c_img, bx, by, ex, ey, bx+(m_boxw/4), by+(m_boxh/4),
  ex-(m_boxw/4),ey-(m_boxh/4), null);
 g.setColor(Color.white);
 g.drawRect(bx, by, m_boxw, m_boxh);

}

public void mouseDragged(MouseEvent e) {
 drawZoomBox(e);
}

public void mouseMoved(MouseEvent e) {

 Graphics g = getGraphics();
 if (m_pressed && (m_button == 1)) {
  drawZoomBox(e);
  g.setColor(Color.white);
  g.drawString("DRAG", 10,10);
 } else {

  g.setColor(Color.white);
  String s = "("+e.getX()+","+e.getY()+")";
  repaint();
  g.drawString(s, e.getX(), e.getY());
 }

}

public void mouseClicked(MouseEvent e) {}
public void mouseEntered(MouseEvent e) {}
public void mouseExited(MouseEvent e) {}

public void mousePressed(MouseEvent e) {
 m_pressed = true;
 m_button = e.getButton();
 repaint();
 if (m_button == 1) drawZoomBox(e);
}

public void mouseReleased(MouseEvent e) {
 m_pressed = false;
 m_button = 0;
}
```

```
}
```

And copy and paste the following text into a file, then save it as `RSIImageAreaResizeListener.java`:

```
public interface RSIImageAreaResizeListener {
 public void areaResized(int newx, int newy);
}
```

Compile these classes in Java. Then, either update the `jbexamples.jar` file in the `external/objbridge/java` directory with the new compiled class, place the resulting compiled classes in your Java class path, or edit the JVM Classpath setting in the IDL-Java bridge configuration file to specify the location (path) of these compiled classes. See “Configuring the Bridge” on page 274 for more information.

With the Java classes compiled, you can now access them in IDL. Copy and paste the following text into the IDL Editor window, then save it as `ImageFromJava.pro`:

```
PRO ImageFromJava
; Create a Swing Java object and have it load image data
; into IDL.

; Create the Java object first.
oJSwing = OBJ_NEW('IDLjavaObject$FrameTest', 'FrameTest')

; Get the image from the Java object.
image = oJSwing -> GetImageData()
PRINT, 'Loaded Image Information:'
HELP, image

; Delete the Java object.
OBJ_DESTROY, oJSwing

; Interactively display the image.
IIMAGE, image

END
```

After compiling the above routine, you can run it in IDL. This routine produces the following Java Swing application.

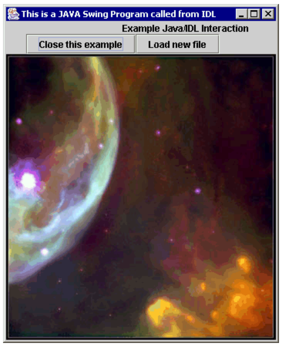

This is a JAVA Swing Program called from IDL
Example Java/IDL Interaction
Close this example
Load new file

Figure 4-2: Java Swing Application Example

Then, the routine produces the following iImage tool.

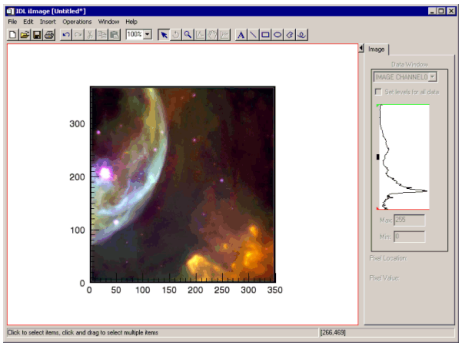


Figure 4-3: iImage Tool from Java Swing Example

Note

After IDL starts the Java Swing application, the two displays are independent of each other. If a new image is loaded into the Java application, the IDL iImage tool is not updated. If the iImage tool modifies the existing image or opens a new image, the Java Swing application is not updated.

Troubleshooting Your Bridge Session

The IDL-Java bridge provides error messages for specific types of operations. These messages can be used to determine when these errors occur, how these errors happen, and what solutions can be applied. The following sections pertain to these error messages and their possible solutions for each type of operation:

- "Errors when Initializing the Bridge"
- "Errors when Creating Objects" on page 315
- "Errors when Calling Methods" on page 316
- "Errors when Accessing Data Members" on page 317

Errors when Initializing the Bridge

The IDL-Java bridge initializes when the first Java object in IDL is created. If the bridge is not configured correctly, an error message is issued and the IDL stops. The following errors occur because the IDL-Java bridge cannot find the Java Virtual Machine on your system. On UNIX, check the $IDLJAVAB_LIB_LOCATION environment variable, and on Windows, check the IDLJAVAB_LIB_LOCATION environment variable. If this environment variable does not exist on your system, create it and set it equal to the location of the Java Virtual Machine on your system. See "Configuring the Bridge" on page 274 for details:

- `Bad JVM Home value: '`*path*`'`, where *path* is the location of Java Virtual Machine on your system.
- *JVM shared lib* `not found in path '`*JVM LibLocation*`'`, where *JVM shared lib* is the location of the Java Virtual Machine shared library and *JVM LibLocation* is the value of the IDLJAVAB_LIB_LOCATION environment variable.
- `No valid JVM shared library exists at location pointed to by $IDLJAVAB_LIB_LOCATION`
- `idljavab.jar not found in path '`*path*`'`, where *path* is the location of the `external/objbridge/java` directory in the IDL distribution.
- `Bridge cannot determine which JVM to run`

- `Java virtual machine failed to start`
- `Failure loading JVM:` *`path/JVM shared lib name`*, where *path* is the location of the Java Virtual Machine and *JVM shared lib name* is the name of the main Java shared library, which is usually `libjvm.so` on UNIX and `jvm.dll` on Windows.

If IDL catches an error and continues, subsequent attempts to call the bridge will generate the following message:

- `IDL-Java bridge is not running`

If this message occurs, fix the error and restart IDL.

Errors when Creating Objects

The following error messages can occur while creating a Java object in IDL. Possible solutions for these errors are also provided:

- `Wrong number of parameters` - occurs if OBJ_NEW does not have 2 or more parameters. Make sure you are specifying the class name twice; once in uppercase with periods replaced by underscores for IDL, and another with periods for Java. See "Java Class Names in IDL" on page 283 for details.
- `Second parameter must be the Java class name` - occurs if 2nd parameter is not an IDL string. When using OBJ_NEW, make sure the Java class name parameter is an IDL string. In other words, the class name has a single quote mark before and after it. See "Java Class Names in IDL" on page 283 for details.
- `Class` *classname* `not found`, where *classname* is the class name you specified in the first two parameters to OBJ_NEW - occurs if the IDL-Java bridge cannot find the class name specified. Check the spelling of each class name parameter and make sure the class name specified for IDL is referring to the same type of object specified for the Java class name. If the parameters are correct, check the Classpath setting in the IDL-Java bridge configuration file. Make sure the Classpath is set to the correct path for the class files containing the *classname* class. See "Configuring the Bridge" on page 274 for details.

- `Class` *classname* `is not a public class`, where *classname* is the class name you specified in the first two parameters to OBJ_NEW - occurs if specified class is not a public class. Edit your Java code to make sure the class you want to access is public.
- `Constructor` *class*`::`*class*`(`*signature*`) not found`, where *class* is the class name - occurs if the IDL-Java bridge cannot find the class constructor with the given parameters. Check the spelling of the specified parameters and look in your Java code to see if you are specifying the correct arguments for the class you are trying to create. Also check to ensure your IDL data can be promoted to the data types in the Java signature. See "Java Class Names in IDL" on page 283 for details.
- `Illegal IDL value in parameter` *n*, where *n* is the position of the parameter - occurs if an illegal parameter type is provided. For example, an IDL structure is not allowed as a parameter to an IDLjavaObject.
- `Exception thrown` - occurs if an exception occurs in Java. Either correct or handle the Java exception. The Java exception can be determined with the IDLJavaBridgeSession object. See "The IDLJavaBridgeSession Object" on page 291 for details.

Errors when Calling Methods

The following error messages can occur while calling methods to Java objects in IDL. Possible solutions for these errors are also provided:

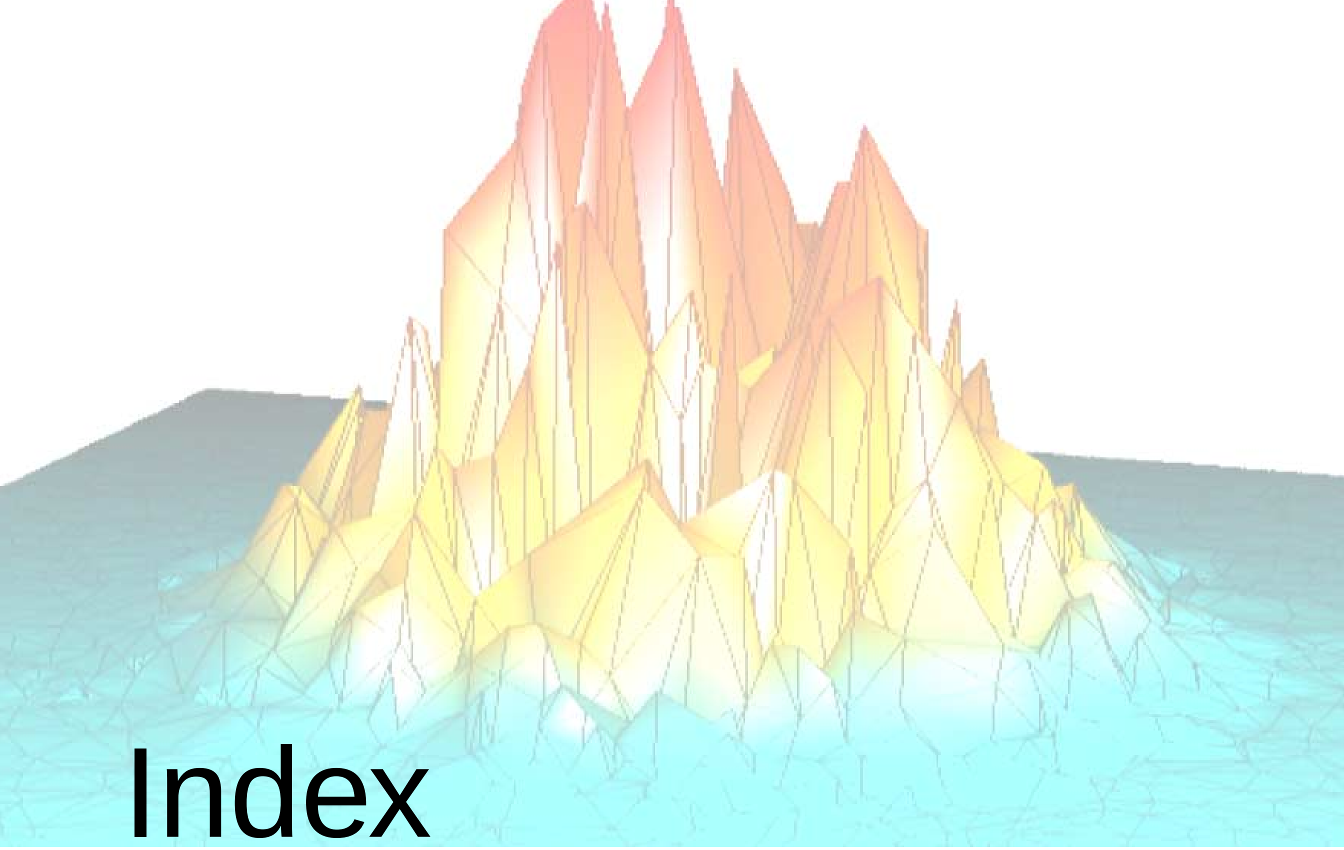
- `Illegal IDL value in parameter` *n*, where *n* is the position of the parameter - occurs if an illegal parameter type is provided. For example, an IDL structure are not allowed as a parameter to an IDLjavaObject.
- `Class` *class* `has no method named` *method*, where *class* is the class name and *method* is the method name specified when trying to call the Java method - occurs if the method of given name does not exist. Check the spelling of the method name. Also compare the method name in the Java class source file with the method name provided when calling the method in IDL. See "What Happens When a Method Call is Made?" on page 285 for details.
- *class*`::`*method*`(`*signature*`) is a void method. Must be called as a procedure`, where *class* is the class name and *method* is the method name specified when a void Java method is called as an IDL function. Change the syntax of the method call. See "Method Calls on IDL-Java Objects" on page 285 for details.

- `Method` *class*`::`*method*`(`*signature*`) not found`, where *class* is the class name and *method* is the method name specified when trying to call the Java method - occurs if the IDL-Java bridge cannot find the method with a matching signature. Check the spelling of the method name. Also compare the method name in the Java class source file with the method name provided when calling the method in IDL. Also check to ensure your IDL data can be promoted to the Java signature. See "What Happens When a Method Call is Made?" on page 285 for details.
- `Exception thrown` - occurs if an exception occurs in Java. Either correct or handle the Java exception. The Java exception can be determined with the IDLJavaBridgeSession object. See "The IDLJavaBridgeSession Object" on page 291 for details.

Errors when Accessing Data Members

The following error messages can occur while accessing data members to Java objects in IDL. Possible solutions for these errors are also provided:

- `Illegal IDL value in parameter` *n*, where *n* is the position of the parameter - occurs if an illegal parameter type is provided. For example, an IDL structure is not allowed as a parameter to an IDLjavaObject.
- `Class` *class* `has no data member named` *property*, where *class* is the class name and *property* is the data member name specified when trying to access the Java data member - occurs if the data member of the given name does not exist. Check the spelling of the property name. Also compare the data member name in the Java class source file with the property name provided when accessing it in IDL. See "Managing IDL-Java Object Properties" on page 287 for details.
- `Property` *class*`::`*property* `of type` *type* `not found`, where *class* is the class name, *property* is the data member name specified, and *type* is *property*'s data type when trying to access the Java data member - occurs if the IDL-Java bridge cannot find the Java data member of the given type. Check the data type of Java data member and make sure you are trying to use a similar type in IDL. See "Getting and Setting Properties" on page 288 for details.
- `Exception thrown` - occurs if an exception occurs in Java. Either correct or handle the Java exception. The Java exception can be determined with the IDLJavaBridgeSession object. See "The IDLJavaBridgeSession Object" on page 291 for details.



Index