IDL®

What's New in IDL 5.6

RESEARCH SYSTEMS
A Kodak Company

IDL Version 5.6
October, 2002 Edition

1002IDL56WN

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.

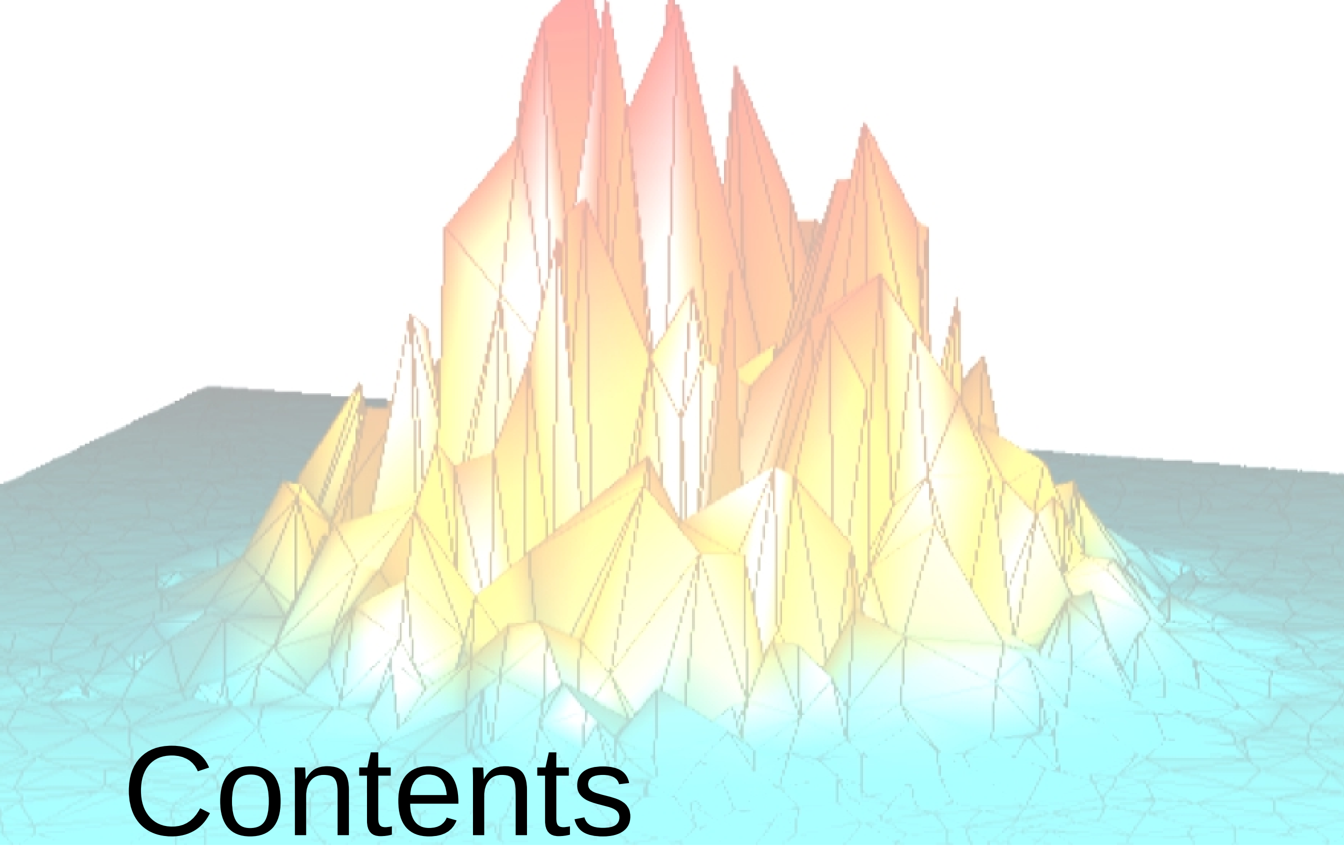
Use of this software for providing LZW capability for any purpose is not authorized unless user first enters into a license agreement with Unisys under U.S. Patent No. 4,558,302 and foreign counterparts. For information concerning licensing, please contact: Unisys Corporation, Welch Licensing Department - 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.

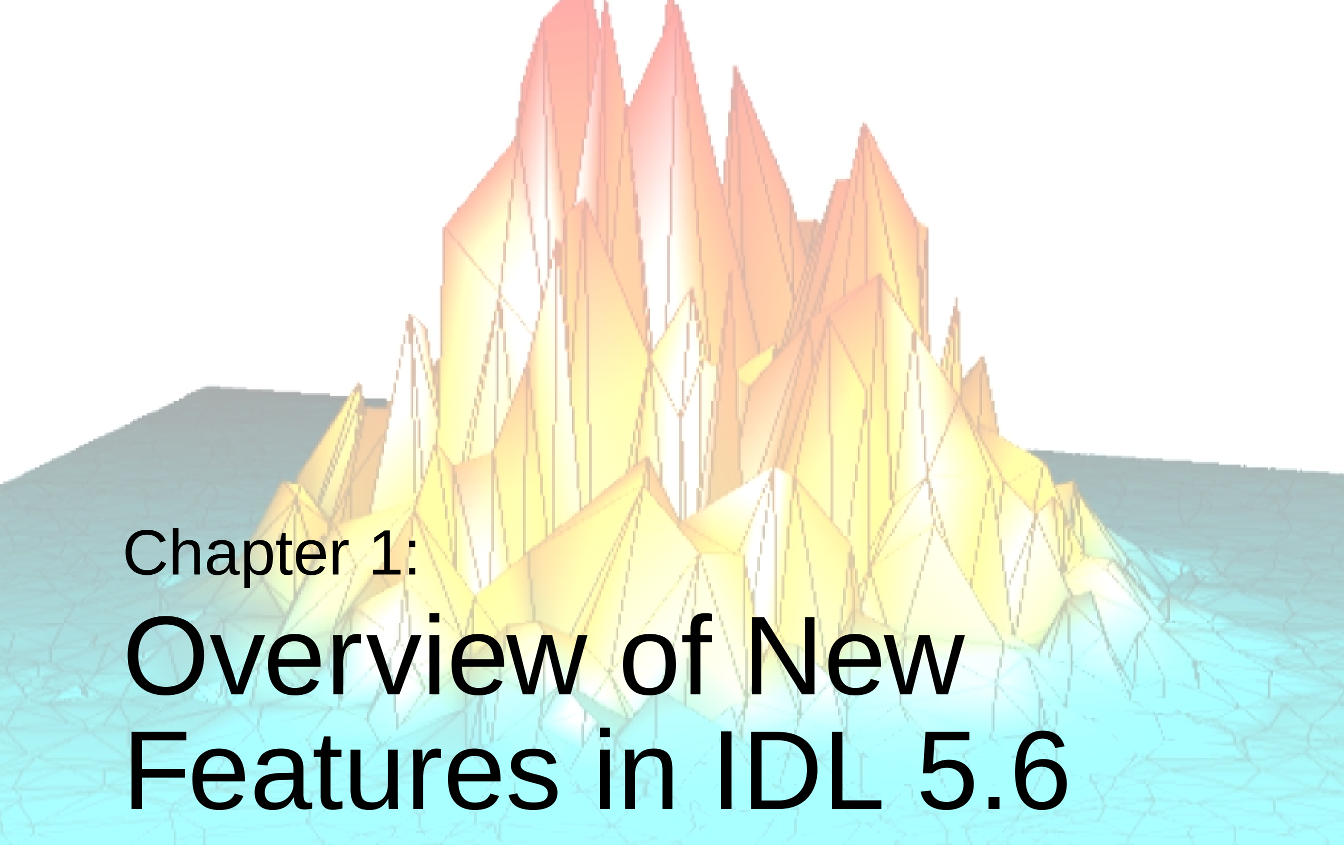
This product includes software developed by the Apache Software Foundation (http://www.apache.org/)

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 5.6

This chapter contains the following topics:

Visualization Enhancements

The following enhancements have been made to IDL's visualization functionality for the 5.6 release:

- Mesa Library Update
- Labels for Contour Objects
- Labels for Polyline Objects
- Labels for ISOCONTOUR
- New User-Defined Clipping Planes for Objects
- New Keyword to Determine the Maximum Number of Clipping Planes
- Enhancements for Displaying Points and Lines in Object Graphics
- OpenGL Hardware Support for Object Graphics on HP and Linux
- New User-Defined Cursor Registration
- New Keyword to PickData Method

Mesa Library Update

IDL 5.6 incorporates a new release of the Mesa library (version 4.0.1). This library is used in IDL Object Graphics displays for platform independent software rendering when the **Object Graphics Software** rendering preference is selected.

The benefits to this update are the following:

- Any graphical output derived when using software rendering in IDL conforms to OpenGL standards since the Mesa 4.0 library passes the OpenGL conformance suite.
- The Tesselator is much more robust, allowing it to tessellate complex and degenerate polygons and TrueType fonts.

For information on the changes to the IDLgrTessellator object due to this upgrade, see "New and Enhanced IDL Objects" on page 60. For information on the issues that have been solved in IDL due to this update, see the IDL release notes.

Labels for Contour Objects

IDL 5.6 now supports labeling of contour lines for the IDLgrContour object.

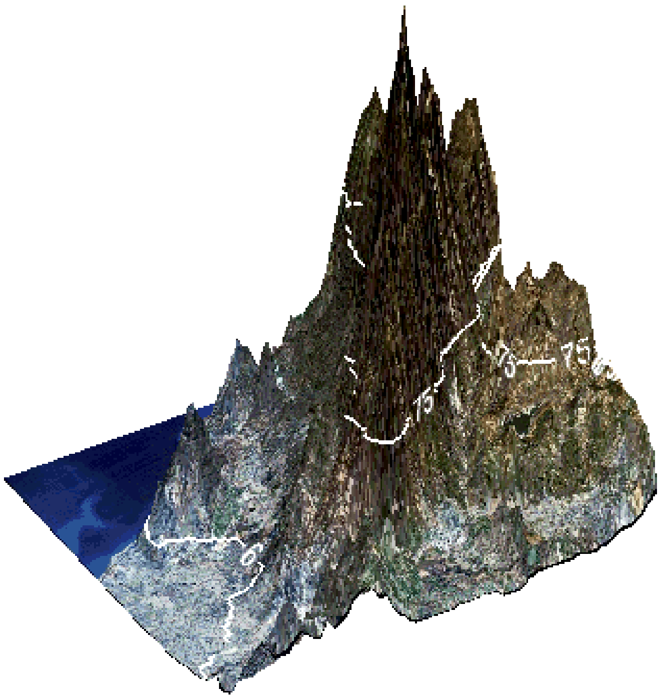


Figure 1-1: New Labeling of Contour LInes Using IDLgrContour::Init

The following changes have been made to the IDLgrContour object class to support this new functionality:

- New keywords to the IDLgrContour::Init method:
    - AM_PM
    - C_LABEL_INTERVAL
    - C_LABEL_OBJECTS
    - C_LABEL_NOGAPS
    - C_LABEL_SHOW
    - C_USE_LABEL_COLOR

    - C_USE_LABEL_ORIENTATION
    - DAYS_OF_WEEK
    - LABEL_FONT
    - LABEL_FORMAT
    - LABEL_FRMTDATA
    - LABEL_UNITS
    - MONTHS
    - USE_TEXT_ALIGNMENTS
- New IDLgrContour::GetLabelInfo method.
- New IDLgrContour::AdjustLabel method.

For a description of the new keywords and methods, see “New and Enhanced IDL Objects” on page 60.

Labels for Polyline Objects

IDL 5.6 now supports the labeling of polyline paths for IDLgrPolyline objects with text and symbol objects. New keywords to the IDLgrPolyline::Init method which support labeling of polyline paths are:

- LABEL_NOGAPS
- LABEL_POLYLINES
- LABEL_OFFSETS
- LABEL_OBJECTS
- LABEL_USE_VERTEX_COLOR
- USE_LABEL_COLOR
- USE_LABEL_ORIENTATION
- USE_TEXT_ALIGNMENTS

For a description of the new keywords, see “IDLgrPolyline::Init” on page 75.

In addition, the *Data* argument to IDLgrSymbol::Init now contains a new pre-defined Arrow-Head (>) symbol (represented by the scalar 8). This allows you to easily add arrowheads to the results of PARTICLE_TRACE.

Labels for ISOCONTOUR

IDL 5.6 now supports the labeling of contour lines for the ISOCONTOUR routine. New keywords to the ISOCONTOUR routine which support labeling of contour lines are:

- C_LABEL_INTERVAL
- C_LABEL_SHOW
- OUT_LABEL_OFFSETS
- OUT_LABEL_POLYLINES
- OUT_LABEL_STRINGS

For a description of the new keywords, see "ISOCONTOUR" on page 115.

New User-Defined Clipping Planes for Objects

IDL 5.6 now adds a new CLIP_PLANES keyword for all atomic graphic object classes and to IDLgrModel. This allows you to specify the coefficients of the clipping planes you wish to be applied to an object and its children, if applicable. Multiple clipping planes can be applied, up to the maximum number supported by the device. The new keyword applies to the Init method of IDLgrAxis, IDLgrContour, IDLgrImage, IDLgrModel, IDLgrPlot, IDLgrPolygon, IDLgrPolyline, IDLgrROI, IDLgrROIGroup, IDLgrSurface, IDLgrText and IDLgrVolume.

For more information, see "IDL Object Method Enhancements" on page 61.

New Keyword to Determine the Maximum Number of Clipping Planes

IDL 5.6 contains a new MAX_NUM_CLIP_PLANES keyword for the GetDeviceInfo method to IDLgrBuffer, IDLgrClipboard, IDLgrWindow, and IDLgrVRML. This keyword returns an integer that specifies the maximum number of user-defined clipping planes supported by the device for use with the new CLIP_PLANES keyword to atomic graphic objects.

For more information, see "IDL Object Method Enhancements" on page 61.

Enhancements for Displaying Points and Lines in Object Graphics

Point and line texture mapping allows you to more accurately control the color of a line or point as well as control the alpha blending of a line or point to achieve transparency effects.

IDL will now automatically render a texture map on an IDLgrPolygon or IDLgrSurface object when:

- A valid IDLgrImage object is specified in the TEXTURE_MAP property.
- One of the following are set for the STYLE property:
  - Points
  - Wire mesh (IDLgrSurface only)
  - Lines (IDLgrPolygon only)
  - Filled
  - RuledXZ (IDLgrSurface only)
  - RuledYZ (IDLgrSurface only)

The following example demonstrates using a texture map on an IDLgrSurface object:

```
PRO ExTextureOnRoad

; Read elevation data file.
filename = FILEPATH('elevbin.dat', $
   SUBDIRECTORY = ['examples', 'data'])
dim = 64
heightField = READ_BINARY(filename, $
   DATA_DIMS = [dim, dim])

; Get the color data from a palette suitable for
; elevation.
oPalette = OBJ_NEW('IDLgrPalette')
oPalette -> LoadCT, 4
oPalette -> GetProperty, RED_VALUES = red, $
   GREEN_VALUES = green, BLUE_VALUES = blue
OBJ_DESTROY, oPalette

; Create the texture.
; Colors correspond to height.
; Set alpha to be transparent where the elevation is
; zero (suggesting water level).
texture = BYTARR(4, dim, dim, /NOZERO)
```

```
texture[0,*,*] = red[heightfield]
texture[1,*,*] = green[heightfield]
texture[2,*,*] = blue[heightfield]
alpha = BYTARR(dim,dim)
above = WHERE(heightField gt 0)
alpha[above] = 255
texture[3,*,*] = alpha
oTexture = OBJ_NEW('IDLgrImage', texture)

; Scale the geometry for better viewing
heightField = BYTSCL(heightField, TOP = 63) + 192

; Create textured surface model.
; Experiment with the STYLE property to see texturing
; effects for different styles.
oSurface = OBJ_NEW('IDLgrSurface', heightField, $
   COLOR = [255, 255, 255], TEXTURE_MAP = oTexture, $
   STYLE = 2)

; Create a "road" that is color coded with the texture
; to indicate elevation. The first (black) line shows
; the intended path of the road. The second line
; (textured and on top) shows the elevation of each
; point of the road, color coded with the texture.  The
; missing sections of this line, where the black shows
; through, indicate where the road passes over water.
road = [[0, 10], [30, 20], [40, 30], [45, 50], $
   [12, 50], [2, 35], [5, 20]]
oLine1 = OBJ_NEW('IDLgrPolygon', road, $
   COLOR = [0, 0, 0], STYLE = 1, THICK = 5, $
   ZCOORD_CONV = [191, 1])
oLine2 = OBJ_NEW('IDLgrPolygon', road, $
   COLOR = [255, 255, 255], TEXTURE_MAP = oTexture, $
   TEXTURE_COORD = FLOAT(road)/dim, STYLE = 1, $
   THICK = 5, ZCOORD_CONV = [192, 1])

XOBJVIEW, [oSurface], /BLOCK, $
   TITLE = 'Texture Map on Surface', $
   XSIZE = 600, YSIZE = 400
XOBJVIEW, [oLine1, oLine2], /BLOCK, $
   TITLE = 'Texture Map on Road Polygon', $
   XSIZE = 600, YSIZE = 400
OBJ_DESTROY, [oTexture, oSurface, oLine1, oLine2]

END
```

OpenGL Hardware Support for Object Graphics on HP and Linux

IDL 5.6 now includes OpenGL hardware support for object graphics on the HP and Linux platforms. OpenGL support is set by default and may be changed from the IDLDE by selecting **File** → **Preferences** → **Graphics**.

Note

On the HP-UX platform, the indexed color model is not supported by the OpenGL hardware accelerator. IDL reverts to software rendering if the color model property of a destination object is set to indexed color (1).

New User-Defined Cursor Registration

IDL 5.6 now supports user-defined cursor registration by means of the new REGISTER_CURSOR procedure. This allows you to define a bitmap to display as the cursor in an IDLgrWindow to indicate the mouse position. Once a new cursor is registered, it is accessible via the IDLgrWindow::SetCurrentCursor method.

For more information see “REGISTER_CURSOR” on page 419 and “IDLgrWindow::SetCurrentCursor” on page 88.

New Keyword to PickData Method

The new PICK_STATUS keyword to the PickData method of the IDLgrBuffer and IDLgrWindow allows you to retrieve information about individual pixels within a *pick* box defined using the DIMENSIONS keyword.

For more information on the PICK_STATUS keyword, see “IDLgrWindow::PickData” on page 87.

Analysis Enhancements

The following enhancements have been made to IDL's data analysis functionality for the 5.6 release:

- New LAPACK Linear Algebra Routines
- New DIAG_MATRIX Function
- New MATRIX_POWER Function
- New PRODUCT Function
- New Run-length Encoding for ROI Masks
- New Complex Input Support
- Enhancements to ATAN
- Enhancements to the BESEL Functions
- Enhancement to the CURVEFIT Function
- Enhancements to the EXPINT Function
- Enhancements to the GAUSSFIT Function
- Enhancements to the MEDIAN Function

New LAPACK Linear Algebra Routines

The LAPACK numerical library has been integrated into IDL to give you more robust and accurate algorithms for solving systems of linear equations, singular value decomposition, solving eigenvalue problems, and for linear least-squares calculations. IDL 5.6 includes 23 new routines that use the included LAPACK linear algebra package. These routines are from the CLAPACK library, based on the FORTRAN LAPACK version 3.0 library. For more information visit the Netlib Repository at `http://www.netlib.org`. For more details see Anderson et al., *LAPACK Users' Guide, 3rd ed.*, SIAM, 1999.

Sixteen of the new LAPACK routines supply similar functionality to existing Numerical Recipes routines. In these cases, the names have been taken from the Numerical Recipes routine, with the addition of the LA_ prefix. Note however that the LAPACK routine may not accept the same arguments or keywords as the Numerical Recipes routine. Also, because of differences in algorithms between LAPACK and Numerical Recipes, the results for the same procedure may have different numerical values or different ordering.

In general, compared to Numerical Recipes, the LAPACK routines are more robust and accurate, may be faster, and also handle complex numbers.

Note

For LAPACK routines that accept arrays, the arrays are expected to be in IDL's column-major format, where the first dimension represents the columns and the second dimension represents the rows. Internally, a transpose is automatically applied before the array is passed to the LAPACK C routine. Likewise, a transpose is applied to the result before it is returned to the user.

The following table lists the available LAPACK routines, along with a short description, Asterisks mark routines that also have a corresponding Numerical Recipes routine in IDL.

Linear Equations

LAPACK Routine	Description
LA_CHOLDC*	Cholesky factorization.
LA_CHOLMPROVE	Improve solution using Cholesky factorization.

Table 1-1: Linear Equation Routines

LAPACK Routine	Description
LA_CHOLSOL*	Solve linear equations using Cholesky.
LA_DETERM*	Array determinant using LU decomposition.
LA_INVERT*	Array inverse using LU decomposition.
LA_LINEAR_EQUATION	Solve linear equations using LU decomposition.
LA_LUDC*	LU decomposition.
LA_LUMPROVE*	Improve solution using LU decomposition.
LA_LUSOL*	Solve linear equations using LU decomposition.
LA_TRIDC	LU decomposition of tridiagonal array.
LA_TRIMPROVE	Improve solution of a tridiagonal problem.
LA_TRISOL*	Solve linear equations with a tridiagonal array.

Table 1-1: Linear Equation Routines

Note

* Has a corresponding Numerical Recipes routine.

Eigenvalues and Eigenvectors

LAPACK Routine	Description
LA_EIGENPROBLEM	Eigenvalues and eigenvectors with error estimates for nonsymmetric arrays.
LA_EIGENQL*	Compute selected eigenvalues and eigenvectors for symmetric or Hermitian array.
LA_EIGENVEC*	Selected eigenvectors of nonsymmetric array.

Table 1-2: Eigenvalue and Eigenvector Routines

LAPACK Routine	Description
LA_ELMHES*	Reduce nonsymmetric array to upper Hessenberg.
LA_HQR*	Eigenvalues of upper Hessenberg.
LA_TRIQL*	Eigenvalues and eigenvectors of tridiagonal array.
LA_TRIRED*	Reduce symmetric array to tridiagonal.

Table 1-2: Eigenvalue and Eigenvector Routines

Note

* Has a corresponding Numerical Recipes routine.

Linear Least Squares

LAPACK Routine	Description
LA_GM_LINEAR_MODEL	Solve general Gauss-Markov linear model.
LA_LEAST_SQUARE_EQUALITY	Solve linear least-squares problem with constraint.
LA_LEAST_SQUARES	Solve linear least-squares problem.

Table 1-3: Linear Least Square Routines

Singular Value Decomposition

LAPACK Routine	Description
LA_SVD*	Singular value decomposition.

Table 1-4: Singular Value Decompositon Routine

Note

* The corresponding Numerical Recipes routine is SVDC.

For more information on these new routines, see Chapter 3, “New IDL Routines”.

New DIAG_MATRIX Function

The new DIAG_MATRIX function constructs a diagonal matrix from an input vector, or if given a matrix, then DIAG_MATRIX will extract a diagonal vector.

For more details, see "DIAG_MATRIX" on page 193.

New MATRIX_POWER Function

The new MATRIX_POWER function computes the product of a matrix with itself. For example, the fifth power of array *A* is *A* # *A* # *A* # *A* # *A*. Negative powers are computed using the matrix inverse of the positive power.

The result is a square array containing the value of the matrix raised to the specified power. A power of zero returns the identity matrix.

For more details, see "MATRIX_POWER" on page 414.

New PRODUCT Function

The new PRODUCT function returns the product of elements within an array. The product of the array elements can also be computed over a given dimension. This new routine is similar to the TOTAL function used to sum elements within an array.

For more details, see "PRODUCT" on page 416.

New Run-length Encoding for ROI Masks

In IDL 5.6, you can now return a run-length encoded result for an ROI mask. For large ROIs, a considerable savings in space results. With the new RUN_LENGTH keyword set for IDLanROI:ComputeMask or IDLanROIGroup::ComputeMask methods, the ROI mask result is a vector wherein each even-numbered subscript contains the length of the run, and the following element contains the starting index of the run.

For a description of the new keywords, see "IDLanROI::ComputeMask" on page 61 and "IDLanROIGroup::ComputeMask" on page 61.

New Complex Input Support

IDL 5.6 now supports complex input arguments for the following routines; GAMMA, LNGAMMA, BETA, IBETA, IGAMMA, ERF, ERFC, and ERFCX.

For more information, see "IDL Routine Enhancements" on page 111.

Enhancements to ATAN

The new PHASE keyword to the ATAN function can be used to compute ATAN (Imaginary(Z), Real_part(Z)), but uses less memory and is faster. This new keyword restores functionality provided by ATAN in IDL versions prior to 5.5.

For more information see “ATAN” on page 111.

Enhancements to the BESEL Functions

The new DOUBLE keyword to IDL’s Bessel functions allows you to specify whether the functions should return a single- or a double-precision result.

The new ITER keyword to IDL’s Bessel functions allows you to retrieve the maximum number of iterations for which the function will converge for a given value.

For more information on these new keywords, see “BESELI, BESELJ, BESELK, BESELY” on page 111.

Enhancement to the CURVEFIT Function

The new YERROR keyword to CURVEFIT can be set to a named variable that will contain the standard error between YFIT and Y.

For more information on this new keyword, see “CURVEFIT” on page 112.

Enhancements to the EXPINT Function

The new ITER keyword to EXPINT defines a named variable that will contain the actual number of iterations performed.

For more information on this new keyword, see “EXPINT” on page 113.

Enhancements to the GAUSSFIT Function

The following enhancements have been made to the GAUSSFIT function:

- The new YERROR keyword to GAUSSFIT returns the error associated with the fit.
- The new SIGMA keyword to GAUSSFIT can be set to a named variable that will contain the 1-sigma error estimates of a returned parameters.

- The new CHISQ keyword to GAUSSFIT can be set to a named variable that will contain the value of the chi-square goodness-of-fit.

For more information on these new keywords, see "GAUSSFIT" on page 114.

Enhancements to the MEDIAN Function

The new DIMENSION keyword to MEDIAN can be set to the dimension over which to find the median values of an array.

For more information on this new keyword, see "MEDIAN" on page 117.

Language Enhancements

The following enhancements have been made to the core of the IDL Language for the 5.6 release:

- New Stride Syntax for Array Subscripts
- New Shared Memory Support
- New and Enhanced File Handling Routines
- New SWAP_ENDIAN_INPLACE Procedure
- New Keywords to SWAP_ENDIAN Function
- Enhancements to the EXPAND_PATH Function
- Enhancements to the MAKE_DLL Procedure
- New STRICTARRSUBS Option to COMPILE_OPT
- Large File Support for AIX and Linux Platforms
- Large File Support For Compressed Files
- 64-bit Memory Support On More Platforms
- Thread Pool and Multi-Threading Support On AIX and Mac OS X
- Enhancements to the KEYWORD_SET Function

New Stride Syntax for Array Subscripts

You can now simplify your coding by specifying array subscript ranges using strides, or subscripting increments. The syntax [e_0:e_1:e_2] denotes every e_2th element within the range of subscripts e_0 through e_1 (e_0 must not be greater than e_1). e_2 is referred to as the subscript *stride*. The stride value must be greater than or equal to 1. If it is set to the value 1, the resulting subscript expression is identical in meaning to [e_0:e_1], as described above.

For example, if the variable VEC is a 50-element vector, VEC[5:13:2] is a five-element vector composed of VEC[5], VEC[7], VEC[9], VEC[11], and VEC[13].

The following table summarizes the possible forms of subscript ranges:

Form	Description
e	A simple subscript expression
$e_0:e_1$	Subscript range from e_0 to e_1
$e_0:e_1:e_2$	Subscript range from e_0 to e_1 with a stride of e_2
$e_0:*$	All points from element e_0 to end
$e_0:*:e_2$	All points from element e_0 to end with a stride of e_2
*	All points in the dimension

Table 1-5: Subscript Ranges

For additional information, see Chapter 6, “Arrays” in the *Building IDL Applications* manual.

New Shared Memory Support

Four new IDL routines allow you to map anonymous shared memory, or local disk files, into the memory address space of the currently executing IDL process. Mapped memory segments are associated with an IDL array specified by the user.

Additionally, the new SHARED_MEMORY keyword to the HELP procedure can be used to display information about all current shared memory and memory mapped file segments mapped into the current IDL process via the SHMMAP procedure.

Shared Memory Routine	Description
SHMDEBUG	Debugs shared memory problems.
SHMMAP	Maps memory or disk files into IDL’s memory address space.
SHMUNMAP	Unmaps memory mapped with SHMMAP.
SHMVAR	Creates an IDL array variable that uses memory from a mapped memory segment.

Table 1-6: Routines for Shared Memory Support

Warning

Unlike most IDL functionality, incorrect use of the shared memory routines can corrupt or even crash your IDL process. Proper use of these low level operating system features requires systems programming experience, and is not recommended for those without such experience. You should be familiar with the memory and file mapping features of your operating system and the terminology used to describe such features.

For more information, see "New IDL Routines" on page 89.

New and Enhanced File Handling Routines

New file handling routines in IDL 5.6 further enhance your ability to manipulate files from within IDL.

Routine	Description
COPY_LUN	Copies data between two open files.
FILE_COPY	Copies files, or directories of files, to a new location.
FILE_LINES	Returns the number of lines of text contained within the specified file or files.
FILE_LINK	Creates UNIX file links, both regular (hard) and symbolic.
FILE_MOVE	Renames files and directories.
FILE_READLINK	Returns the path pointed to by UNIX symbolic links.
FILE_SAME	Determines if two different file names refer to the same underlying file.
SKIP_LUN	Reads data in an open file and moves the file pointer.
TRUNCATE_LUN	Truncates the contents of a file open for write access at the current position of the file pointer.

Table 1-7: New IDL File Handling Routines

The FILE_DELETE procedure has been enhanced to allow for more control with error reporting. Two new keywords have been added:

- ALLOW_NONEXISTENT — Quietly ignores attempts to delete a non-existent file.
- VERBOSE — Issues informative messages for every file deleted.

For more information, see “New IDL Routines” on page 89.

New SWAP_ENDIAN_INPLACE Procedure

The new SWAP_ENDIAN_INPLACE procedure reverses the byte ordering of arbitrary scalars, arrays or structures. It can make “big endian” numbers “little endian” and vice-versa.

Note

The BYTEORDER procedure can be used to reverse the byte ordering of *scalars and arrays* (SWAP_ENDIAN_INPLACE also allows structures).

For more information, see “SWAP_ENDIAN_INPLACE” on page 447.

New Keywords to SWAP_ENDIAN Function

Two new keywords have been added to the SWAP_ENDIAN function: SWAP_IF_BIG_ENDIAN and SWAP_IF_LITTLE_ENDIAN. These keywords add the functionality available in the BYTEORDER routine to SWAP_ENDIAN.

For more information, see “SWAP_ENDIAN” on page 118.

Enhancements to the EXPAND_PATH Function

The EXPAND_PATH function has been enhanced to use the same expansion of the path-definition string as is done by IDL when initializing the !PATH system variable from the IDL_PATH environment variable at startup. This means that in addition to expanding the “+” syntax in the path definition string, EXPAND_PATH also properly expands the special path-definition tokens `<IDL_DEFAULT>`, `<IDL_BIN_DIRNAME>`, and `<IDL_VERSION_DIRNAME>`.

Note

This functionality has also been added to the way path preferences are set in the IDL Development Environment. See “Changes to Path Preferences” on page 39 for details.

Enhancements to the MAKE_DLL Procedure

If the new REUSE_EXISTING keyword to the MAKE_DLL procedure is set, and the sharable library file specified by *OutputFile* already exists, MAKE_DLL returns without building the sharable library again. Use this keyword in situations where you wish to ensure that a library exists, but only want to build it if it does not. Combining the REUSE_EXISTING and DLL_PATH keywords allows you to get a path to the library in a platform independent manner, building the library only if necessary.

For more information on this keyword, see "MAKE_DLL" on page 117.

New STRICTARRSUBS Option to COMPILE_OPT

The STRICTARRSUBS option has been added to the COMPILE_OPT statement. When IDL subscripts one array using another array as the source of array indices, the default behavior is to clip any out-of-range indices into range and then quietly use the resulting data without error. This behavior is described in "Array Subscripts" in Chapter 6 of the *Building IDL Applications* manual. Specifying STRICTARRSUBS will instead cause IDL to treat such out-of-range array subscripts within the body of the routine containing the COMPILE_OPT statement as an error. The position of the STRICTARRSUBS option within the module is not important: all subscripting operations within the entire body of the specified routine will be treated this way.

Large File Support for AIX and Linux Platforms

IDL 5.6 now supports accessing files larger than 2.1 GB on AIX and Linux. You now can use the 64-bit integer data type to read and write data from files on the following platforms that support the use of a large file capable file system:

- Windows (with NTFS file system)
- AIX
- Linux
- SUN Solaris
- HP-UX
- SGI Irix
- Compaq Tru64 UNIX

IDL sets the !VERSION.FILE_OFFSET_BITS system variable to 64 on platforms where it has large file support.

Note for AIX Users

Customers attempting to use large file functionality under AIX need to be aware of the following:

- By default, AIX imposes a file size limit of 2097151 512-byte blocks on all processes. This will limit the size of files you can access. One solution is to set the fsize parameter in `/etc/security/limits` to a larger value, or -1 to remove the limit entirely. Users will have to log out and back in to see the benefit of this change.
- By default, local AIX filesystems are not large file capable, and will refuse to hold files larger than 2.1GB in length. This per-filesystem attribute is set when the filesystem is created, and cannot be changed without destroying and re-creating it.

Large File Support For Compressed Files

On platforms that support large files (!VERSION.FILE_OFFSET_BITS is 64), IDL's support for compressed files (COMPRESS keyword to OPEN or SAVE procedures) is now able to read and write compressed files of any length. In previous releases, IDL's support for such files was 32-bit limited.

64-bit Memory Support On More Platforms

A 64-bit program is a program that uses 64-bit memory addresses. Such a program has the ability to access extremely large amounts of memory, well beyond the 2.1GB barrier that exists for 32-bit programs. Previous to this release, 64-bit versions of IDL were supported on the Solaris/Sparc and Compaq Tru64 UNIX platforms. Other versions of IDL were built as 32-bit programs and were limited to the 32-bit memory address that implies. With IDL 5.6, the IBM AIX, SGI IRIX, and HP-UX versions are also available in both 32- and 64-bit form (similar to Solaris/Sparc IDL which comes in both 32- and 64-bit versions).

Thread Pool and Multi-Threading Support On AIX and Mac OS X

Support for the IDL Thread Pool, which was first released in IDL 5.5, is now supported on the AIX platform. It is also supported on the new Mac OS X platform. The Thread Pool is now supported on all IDL platforms.

See Chapter 15, “Multithreading in IDL” in the *Building IDL Applications* manual for more information.

Enhancements to the KEYWORD_SET Function

The KEYWORD_SET function returns true if its argument is defined and is nonzero, and false (0) otherwise. The specific rules by which the value is determined are given in the *IDL Reference Guide*. With IDL 5.6, there has been a small change to these rules, designed to make KEYWORD_SET useful in a larger number of cases. Previously, KEYWORD_SET would return true if it's argument was an array, regardless of the value. This behavior has been changed: Arrays with more than 1 element are treated as before, but 1-element arrays are treated in the same way as scalar arguments, and the value returned by KEYWORD_SET depends on the value of the element.

File Access Enhancements

The following enhancements have been made in the area of File Access in the IDL 5.6 release:

- New Support for ITIFF
- New XML Parser Object
- New HDF5 Routines
- New H5_BROWSER Routine
- HDF and HDF-EOS Library Updates
- Enhanced Support for Shapefiles

New Support for ITIFF

IDL 5.6 now supports reading and writing TIFF files containing JPEG compression (ITIFF). The READ_TIFF, WRITE_TIFF, and QUERY_TIFF routines now support ITIFF. A new option has been added to the COMPRESSION keyword to the WRITE_TIFF routine to support the creation of ITIFF files.

For more information, see "WRITE_TIFF" on page 133.

New XML Parser Object

XML is "eXtensible Markup Language," a popular standard for sharing data across networks and on the World Wide Web. In IDL 5.6, the new IDLffXMLSAX object class implements a SAX 2 XML parsing engine, ideal for extracting data from large XML files. The new object class allows you to extract data from XML data files and store it in IDL data structures. See "IDLffXMLSAX object" on page 146 for a description of the object class and its methods.

Using the XML parser requires that you write a custom subclass of the IDLffXMLSAX object class, overriding the superclass's method routines to read a given XML file and store the data as necessary. See Chapter 4, "Using the XML Parser Object Class" for a detailed description of how to use the XML parser object class.

New HDF5 Routines

You can now query and read HDF5 files in IDL. This hierarchical data storage format was developed by the NCSA to address limitations in HDF4. Several widely-used data products are expected to be distributed in the HDF5 format, including data from NASA's EOS Aura satellite. IDL continues to support HDF4 as well.

The new HDF5 library of routines in IDL 5.6 are in a dynamically-loadable module (DLM) that provides access to the HDF5 library.

The IDL HDF5 library contains the following function categories:

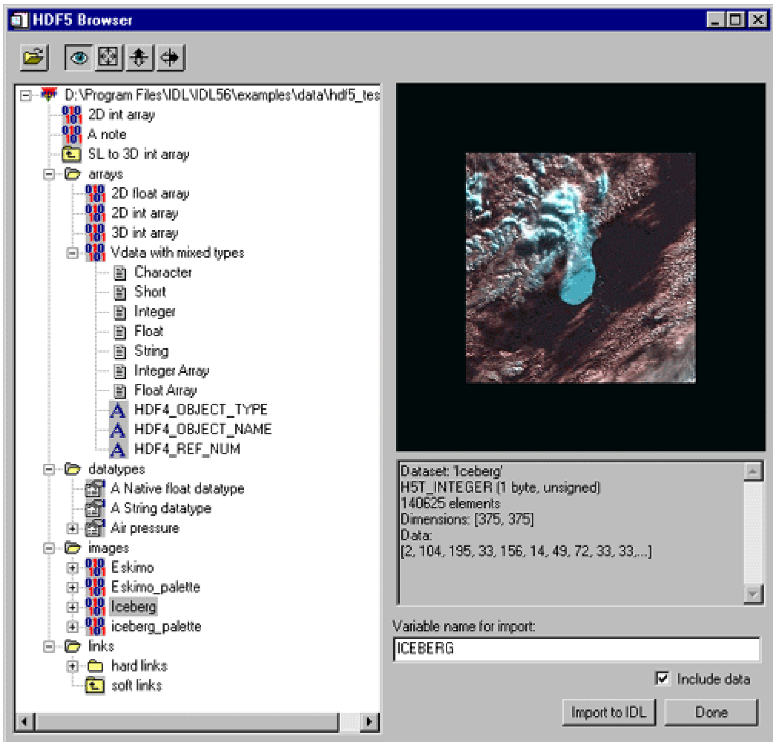
Prefix	Category	Purpose
H5	Library	General library tasks
H5A	Attribute	Manipulate attribute datasets
H5D	Dataset	Manipulate general datasets
H5F	File	Create, open, and close files
H5G	Group	Handle groups of other groups or datasets
H5I	Identifier	Query object identifiers
H5R	Reference	Reference identifiers
H5S	Dataspace	Handle dataspace dimensions and selection
H5T	Datatype	Handle datatype element information

Table 1-8: HDF 5 Function Categories

For more information, see Chapter 3, “New IDL Routines”.

New H5_BROWSER Routine

The new H5_BROWSER presents a graphical user interface for viewing and reading HDF5 files. The browser provides a tree view of the HDF5 file or files, a data preview window, and an information window for the selected objects. The browser may be created as either a selection dialog with **Open**/**Cancel** buttons, or as a standalone browser that can import data to the IDL main program level.



```
Dataset: 'Iceberg'
H5T_INTEGER (1 byte, unsigned)
140625 elements
Dimensions: [375, 375]
Data:
[2, 104, 195, 33, 156, 14, 49, 72, 33, 33,...]
```

Variable name for import: ICEBERG

[✓] Include data

Figure 1-2: The New HDF 5 Browser

For more information, see "H5_BROWSER" on page 214.

HDF and HDF-EOS Library Updates

IDL 5.6 now supports the current versions of the HDF4 and HDF-EOS libraries. HDF is now supported to version 4.1r5 and HDF-EOS is now supported to version 2.8.

Note

On the AIX platform, the HDF and HDF-EOS libraries were not updated. They remain at 4.1r3 for HDF and 2.4.

Enhanced Support for Shapefiles

IDL 5.6 now allows you to access the dBASE table (`.dbf`) component of a shapefile without opening any other components of the shapefile.

For more information, see “IDL Object Method Enhancements” on page 61.

Mapping Enhancements

The list of map projection types available in IDL has been greatly expanded with the addition of the USGS General Cartographic Transformation Package. New routines allow you to set up projections, transform coordinates between projections, and split and clip polygons and polylines to fit your map.

MAP_PROJ_INIT Function

The MAP_PROJ_INIT function establishes the coordinate conversion mechanism for mapping points on a globe's surface to points on a plane, according to either one of the IDL projections or one of the General Cartographic Transformation Package (GCTP) map projections. Unlike MAP_SET, this function does not modify the !MAP system variable, but rather returns a !MAP structure variable that can be used by the map transformation functions MAP_PROJ_FORWARD and MAP_PROJ_INVERSE.

For more information, see "MAP_PROJ_INIT" on page 396.

MAP_PROJ_FORWARD, and MAP_PROJ_INVERSE Functions

These functions transform map coordinates between latitude/longitude and Cartesian (X, Y) coordinates. Both functions can use the map transformation values from either the !MAP system variable or a !MAP structure created by MAP_PROJ_INIT.

See "MAP_PROJ_FORWARD" on page 391 and "MAP_PROJ_INVERSE" on page 412 for details.

IDLDE Enhancements

The IDL Development Environment has been enhanced in the following ways for the 5.6 release:

- Copying and Pasting Multiple IDL Code Lines
- Block Comments
- Changes to Path Preferences

Copying and Pasting Multiple IDL Code Lines

IDL 5.6 for Windows now offers the ability to paste multiple lines of text from the clipboard to the command line. This functionality has been available in previous versions for the UNIX IDLDE. The multi-line command line paste functionality is simple to use. As with the earlier IDLDE, the user needs to merely place some text in the clipboard and paste it into the command line. Any source of text is valid, with emphasis on the requirement that the text be convertible to ASCII. When copying text from an IDE editor, the selection mode can be stream, line, or box.

Note

Line and box modes automatically put a trailing carriage return at the end of the text. When pasted, the last line is executed.

The new functionality should only be used with IDL commands that are contained on one line each, which includes statements that utilize continuation markers ($). Multi-line statements will produce unintended IDL interpreter behavior or errors.

Lines are transferred to the command line as is. Namely, leading white space is not removed and comment lines are sent to the IDL interpreter without distinction.

Note

Tabs are converted to white space based on the tab size indicated by the IDE editor preferences.

Block Comments

In IDL 5.6, the text editor in the IDLDE has been improved to allow quick commenting and uncommenting of code blocks. Previously, comments had to be manually entered using the comment symbol, line by line. To comment lines of code, you may either select the entire block of uncommented lines to be commented or you may simply places the cursor somewhere on the desired line. Commenting and uncommenting can be performed using:

- The **Edit** → **Comment** or **Edit** → **Uncomment** menu items
- The IDL Editor window's context menu
- The toolbar

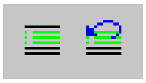


Figure 1-3: Comment (left) and Uncomment (right) Toolbar Icons

Changes to Path Preferences

The path preferences mechanism used by the IDLDE has been modified in this release.

The IDLDE **Path Preferences** dialog now uses the same mechanism to expand the elements of the **IDL File Search Path** field as is used by the EXPAND_PATH function. By default, this field is populated with a single entry: `<IDL_DEFAULT>`. If the IDL_PATH environment variable is *not* set when the IDLDE starts up, it will expand this token into the default value of the !PATH system variable.

Note

If you have set the IDL_PATH environment variable, IDL will set the !PATH system variable based on the contents of the IDL_PATH environment variable at startup, overriding any settings made in the **Path Preferences** dialog. However, after IDL has started, you can modify the current value of the !PATH system variable using this dialog. See "!PATH" in the *IDL Reference Guide* manual for additional details on how !PATH is set.

Setting Path Preferences

If the box to the left of a path element in the **Path Preferences** dialog is checked, all directories below the listed directory that contain at least one `.pro` or `.sav` file will be included in the !PATH system variable. (This mechanism is analogous to the use of a “+” symbol in an EXPAND_PATH path definition string.)

Note

If the `<IDL_DEFAULT>` entry is present, the box to its left is greyed out, indicating that the token will always be expanded.

You can modify the value of the !PATH system variable in the following ways using this dialog:

- **Change the order of the path elements** — using the up- and down-arrows, you can reorder the path elements. When searching the directories in the !PATH system variable for files, IDL will use the first matching file it finds. If you have multiple files with the same name in different directories within !PATH, you may need to adjust the order in which the directories are scanned.
- **Insert...** — To add a path to the IDL **Files Search Path** list, click on **Insert...** to display the **Select Path** dialog. The new path is inserted before the first selected path. If none of the paths are selected, the new path is appended to the end of the list.
- **Insert Standard Libraries** — Click **Insert Standard Libraries** to insert the `<IDL_DEFAULT>` path element into the list.
- **Remove** — Click on **Remove** to delete the selected path.
- **Expand** — Click on **Expand** to include the individual subdirectories of the selected path element in the **Files Search Path** list. When you click **Expand**, the checkmark is removed from the original path element, since the subdirectories are now explicitly included in the path search list.

IDL GUIBuilder Enhancements

The following enhancements have been made to the IDL GUIBuilder in IDL 5.6. For more information on how to use these new features, see Chapter 23, “Using the IDL GUIBuilder” in the *Building IDL Applications* manual.

Support for Tab Widget

The new tab widget is available for inclusion in interfaces built with the GUIBuilder.

Support for Tree Widget

The new tree widget is available for inclusion in interfaces built with the GUIBuilder.

Support for Context Events

Event handlers for context events (triggered when the user clicks the right-hand mouse button) have been added to the property sheets for the base, list, text, and tree widgets.

Support for Tooltips

Support for tooltips has been added to the property sheets for the button and draw widgets.

Support for Checked Menu Items

Support for checked menu items has been added to the menu editor.

Support for Sunken Labels

Support for sunken labels has been added to the property sheet for the label widget.

Support for Move, Iconify, and Size Events for Base Widgets

Support for move, iconify, and size events has been added to the property sheet for the base widget.

Support for Keyboard Events for Draw Widget

Support for keyboard events has been added to the property sheet for the draw widget.

User Interface Toolkit Enhancements

The following enhancements have been made to IDL's UI toolkit for the 5.6 release to help you give your IDL applications more powerful and friendly user interfaces:

- New COM Functionality
- New Combobox Widget
- New Tab Widget
- New Tree Widget
- Table Widget Enhancements
- Move, Iconify, Size Events for Base Widgets
- Color Bitmap Buttons from Array Data
- Push and Toggle Buttons
- Checkmarks on Menu Buttons
- Tooltips for Button and Draw Widgets
- Keyboard Events for Draw Widgets
- Scrolling Draw Widget Enhancements
- Label Widget Enhancements
- Enhancements to WIDGET_INFO

New COM Functionality

The IDLcomIDispatch object class and its use are now described in detail in "Using COM Objects in IDL" in Chapter 4 of the *External Development Guide* manual. The IDLcomIDispatch object has been enhanced in the following ways:

Support for Optional Arguments

Like IDL routines, COM object methods can have *optional arguments*. Optional arguments eliminate the need for the calling program to provide input data for all possible arguments to the method for each call. The COM optional argument functionality is now passed along to COM object methods called on IDLcomIDispatch objects, and to the IDLcomIDispatch::GetProperty method. This means that if an argument is not required by the underlying COM object method, it can be omitted from the method call used on the IDLcomIDispatch object.

Support for Default Values

COM allows objects to specify a default value for any method arguments that are optional. If a call to a method that has an optional argument with a default value omits the optional argument, the default value is used. IDL now behaves in the same way as COM when calling COM object methods on IDLcomIDispatch objects, and when calling the IDLcomIDispatch::GetProperty method.

Support for Argument Skipping

COM allows methods with optional arguments to accept a subset of the full argument list by specifying which arguments are not present. This allows the calling routine to supply, for example, the first and third arguments to a method, but not the second. IDL now provides the same functionality for COM object methods called on IDLcomIDispatch objects, but not for the IDLcomIDispatch::GetProperty or SetProperty methods.

Support for Function Return Values

The original IDL COM subsystem managed function return values through the use of an extra parameter added to the method call. With the addition of optional arguments, this method for retrieving return values is no longer valid. IDL now handles function return values from COM function methods in the same way as IDL functions, using the syntax:

```
Result = oCOM->Method(arg1, arg2)
```

Additional COM Type Mappings

Support for the following COM data types has been added:

COM Type	IDL Type
BOOL (VT_BOOL)	Byte (true =1, false=0)
ERROR (VT_ERROR)	Long
CY (VT_CY)	Long64
DATE (VT_DATE)	Double
I1 (VT_I1)	Byte
INT (VT_INT)	Long
UINT (VT_UINT)	Unsigned Long
VT_USERDEFINED	The IDL type is passed through.

Table 1-9: New IDL-COM Data Type Mappings

New Combobox Widget

The new WIDGET_COMBOBOX function creates comboboxes, which are similar to droplists. The main difference between the combobox widget and the droplist widget is that the combobox widget has an editable text field allowing a value to be entered that is not in the list.

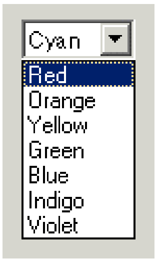


Figure 1-4: Combobox Created Using the New WIDGET_COMBOBOX Function

For more information, see “WIDGET_COMBOBOX” on page 451.

Note

WIDGET_COMBOBOX is not currently available on Compaq True64 UNIX platforms due to that platform's lack of support for the necessary Motif libraries.

New Tab Widget

The new WIDGET_TAB function is used to create a tab widget. Tab widgets present a display area on which different pages (base widgets and their children) can be displayed by selecting the appropriate tab. The titles of the tabs are the values of the TITLE keyword for each of the tag widget's child base widgets.

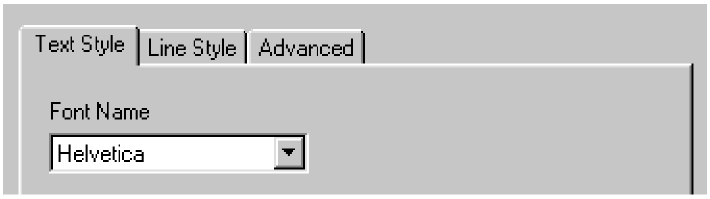


Figure 1-5: Tabs Created Using the New WIDGET_TAB Function

For more information, see "WIDGET_TAB" on page 459.

New Tree Widget

The new WIDGET_TREE function is used to create and populate a tree widget. The tree widget presents a hierarchical view that can be used to organize a wide variety of data structures and information.

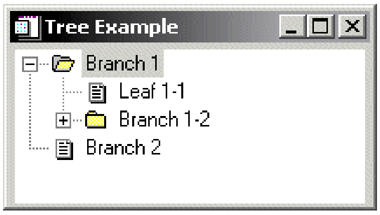


Figure 1-6: A Tree Widget Created Using the New WIDGET_TREE Function

For more information, see "WIDGET_TREE" on page 467.

Table Widget Enhancements

The table widget has been enhanced in the following ways:

Disjoint Cell Selection

Table widgets can now be configured to allow selection of multiple disjoint rectangular groups of cells.

- Use the DISJOINT_SELECTION keyword to WIDGET_TABLE to create a table with this behavior. For more information on this new keyword, see "WIDGET_TABLE" on page 133.
- Use the new TABLE_DISJOINT_SELECTION keyword to WIDGET_CONTROL to change the state of an existing table. For more information on this new keyword, see "WIDGET_CONTROL" on page 121.
- Use the new TABLE_DISJOINT_ SELECTION keyword to WIDGET_INFO to determine the current state of an existing table. For more information on this new keyword, see "WIDGET_INFO" on page 129.

Note

If the USE_TABLE_SELECT keyword to WIDGET_CONTROL is set, the values returned or expected by the following keywords are modified by the selection mode: ALIGNMENT, COLUMN_WIDTHS, DELETE_COLUMNS, DELETE_ROWS, FORMAT, GET_VALUE, ROW_HEIGHTS, and SET_VALUE.

For more on table selection modes, see "Using Table Widgets" in Chapter 26 of the *Building IDL Applications* manual.

New Deselection Event

A new event (TYPE = 9) is generated by the table widget when selected cells are de-selected by the user *and the table is in disjoint selection mode*. This event's structure is identical to the WIDGET_TABLE_CELL_SEL event structure (TYPE = 4) except for the name and type value.

This event occurs when the user holds down the **Control** key when starting a selection and the cell used to start the selection already selected. In contrast, if the user starts a selection with the **Control** key down but starts on a cell that is not selected, the normal WIDGET_TABLE_CELL_SEL event is generated.

```
{WIDGET_TABLE_CELL_DESEL, ID:0L, TOP:0L, HANDLER:0L, TYPE:9,
   SEL_LEFT:0L, SEL_TOP:0L, SEL_RIGHT:0L, SEL_BOTTOM:0L}
```

The range of cells selected is given by the zero-based indices into the table specified by the SEL_LEFT, SEL_TOP, SEL_RIGHT, and SEL_BOTTOM fields.

Cell Selection and Edit Mode

The mechanisms by which an individual table cell is placed in edit mode (that is, made available for interactive editing by the user) have been enhanced to be easier to use. For example, selecting a cell for editing now automatically selects the cell's contents and positions the cursor to the right of the text.

For a complete list of user actions and their effects on cell selection and edit mode, see "Using Table Widgets" in Chapter 26 of the *Building IDL Applications* manual.

Blanking Table Cells

Use the new TABLE_BLANK keyword to WIDGET_CONTROL to cause table cells to be blanked or restored programmatically.

For more information on this new keyword, see "WIDGET_CONTROL" on page 121.

Move, Iconify, Size Events for Base Widgets

Top-level base widgets can now be configured to generate events when the base is moved or iconified. Additionally, an existing base can now be reconfigured to modify the resize event after creation.

For more information on the new keywords described below, see "IDL Routine Enhancements" on page 111.

Move Events

Top-level widget bases return the following event structure when the base is moved and the base was created with the new TLB_MOVE_EVENTS keyword set:

```
{ WIDGET_TLB_MOVE, ID:0L, TOP:0L, HANDLER:0L, X:0L, Y:0L }
```

ID is the widget ID of the base generating the event. TOP is the widget ID of the top level widget containing the base generating the event. HANDLER contains the widget ID of the widget associated with the handler routine. X and Y are the new location of the top left corner of the base.

Note

Move events are generated only when the mouse button is released.

Note

If both TLB_SIZE_EVENTS and TLB_MOVE_EVENTS are enabled, a user resize operation that causes the top left corner of the base to move will generate both a move event and a resize event.

The new TLB_MOVE_ EVENTS keyword to WIDGET_CONTROL allows you to change this setting after the base widget has been created.

The new TLB_MOVE_EVENTS keyword to WIDGET_INFO allows you to determine the current setting.

Iconify Events

Top-level widget bases return the following event structure when the base is iconified or restored and the base was created with the TLB_ICONIFY_EVENTS keyword set:

```
{ WIDGET_TLB_ICONIFY, ID:0L, TOP:0L, HANDLER:0L, ICONIFIED:0 }
```

ID is the widget ID of the base generating the event. TOP is the widget ID of the top level widget containing ID. HANDLER contains the widget ID of the widget associated with the handler routine. ICONIFIED is 1 (one) if the user iconified the base and 0 (zero) if the user restored the base.

The TLB_ICONIFY_ EVENTS keyword to WIDGET_CONTROL allows you to change this setting after the base widget has been created.

The TLB_ICONIFY_EVENTS keyword to WIDGET_INFO allows you to determine the current setting.

Resize Events

In previous releases, you could use the TLB_SIZE_EVENTS keyword to WIDGET_BASE to configure the base widget to generate events when the user resized the widget. IDL 5.6 adds the TLB_SIZE_ EVENTS keyword to WIDGET_CONTROL to allow you to change this setting after the base widget has been created.

The TLB_SIZE_EVENTS keyword to WIDGET_INFO allows you to determine the current setting.

Color Bitmap Buttons from Array Data

In previous versions of IDL, the data for a bitmap to be placed on a button widget could be specified either by setting the VALUE keyword to WIDGET_BUTTON equal to the name of an image file (and specifying the BITMAP keyword) or by setting the VALUE keyword equal to an *n* x *m* array of black-and-white bitmap values. In addition to these two methods, you can now set the VALUE keyword equal to an *n* x *m* x 3 byte array, which displays as a 24-bit color bitmap image.

You can produce appropriate color bitmap arrays in IDL in the following ways:

- Create a 24-bit color image using an external bitmap editor, and read it into an IDL byte array using the appropriate procedure (READ_BMP, READ_JPEG, etc.). The image array must be interleaved by plane (*n* x *m* x 3), with the planes in the order of red, green, and blue.

  Note

  Image files created by image editors are often interleaved by pixel rather than by plane. You can use the TRANSPOSE function to reformat the array.

  ```
  button_image = READ_BMP('bitmap_file.bmp', /RGB)
  button_image = TRANSPOSE(button_image, [1,2,0])
  ...
  button = WIDGET_BUTTON(base, VALUE = button_image)
  ```

- Create an *n* x *m* x 3 byte array using the BYTARR function and modify the array elements using array operations.

Although IDL places no restriction on the size of bitmap allowed, other operating system windowing toolkits IDL interfaces with may prefer certain sizes.

Push and Toggle Buttons

In previous releases, setting the EXCLUSIVE or NONEXCLUSIVE keyword to WIDGET_BASE did not produce the correct visual behavior when the base contained bitmap buttons. In IDL 5.6, bitmap buttons on EXCLUSIVE and NONEXCLUSIVE bases appear selected or unselected in the same manner as buttons with text labels.

In addition, the TOOLBAR keyword to WIDGET_BASE has been added. Setting this keyword does not cause any changes in behavior. Its only affect is to slightly alter the appearance of the bitmap buttons on the base for cosmetic reasons.

On Motif platforms, if bitmap buttons are on a base created with TOOLBAR and either the EXCLUSIVE or NONEXCLUSIVE keywords set, the buttons will not have a separate toggle indicator, they will be grouped closely together, and will have a two-pixel shadow border.

Setting TOOLBAR has no effect on Windows platforms.

Checkmarks on Menu Buttons

Button widgets on menus can now have a checkmark placed next to the button label. Use the CHECKED_MENU keyword to WIDGET_BUTTON to place a check mark next to the button label when the button is created. Use the SET_BUTTON keyword to WIDGET_CONTROL to change the state of the checkmark after creation. Use the BUTTON_SET keyword to WIDGET_INFO to determine the current state of the checkmark on a given button widget.

Note

To be considered a menu button, the button must have as its parent either a button widget created with the MENU keyword or a base widget created with the CONTEXT_MENU keyword.

Tooltips for Button and Draw Widgets

Button and draw widgets can now be configured to display a text tooltip when the mouse cursor hovers over the button or drawable area for a few seconds. Use the TOOLTIP keyword to WIDGET_BUTTON or WIDGET_DRAW to specify the text when the widget is created. Use the TOOLTIP keyword to WIDGET_CONTROL to change the text after the widget has been created. Use the TOOLTIP keyword to WIDGET_INFO to retrieve the current text.

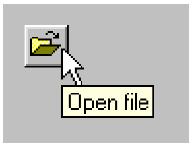


Figure 1-7: A Tooltip With a Button Widget

Keyboard Events for Draw Widgets

Draw widgets can now be configured to generate events when the draw widget has the keyboard focus and a keyboard key is pressed. The event structure returned by the WIDGET_EVENT function is now defined by the following statement:

```
{WIDGET_DRAW, ID:0L, TOP:0L, HANDLER:0L, TYPE: 0, X:0L, Y:0L,
   PRESS:0B, RELEASE:0B, CLICKS:0, MODIFIERS:0L, CH:0, KEY:0L }
```

The TYPE field of the WIDGET_DRAW event structure has been modified to report the following additional values:

Value	Meaning
5	Key Press (ASCII character value reported in CH field)
6	Key Press (Non-ASCII key value reported in KEY field)

Table 1-10: Values to the TYPE Field for WIDGET_DRAW

Keyboard events are generated with the value of the TYPE field equal to 5 or 6. If the event was generated by an ASCII keyboard character, the TYPE field will be set to 5 and the ASCII value of the key will be returned in the CH field. ASCII values can be converted to the string representing the character using the IDL STRING routine. If the event was generated due to a non-ASCII keyboard character, the type of the event will be set to 6 and a numeric value representing the key will be returned in the KEY field. The table below lists the possible values of the KEY field.

Key values reported in the KEY field for the **Shift**, **Control**, **Caps Lock**, and **Alt** keys are not the same as those reported in the MODIFIER field bit mask, since the KEY field is not a bitmask.

Key Field Value	Keyboard Key
1	Shift
2	Control
3	Caps Lock
4	Alt
5	Left

Table 1-11: Key Field Values for the KEY Field for WIDGET_DRAW

Key Field Value	Keyboard Key
6	Right
7	Up
8	Down
9	Page Up
10	Page Down
11	Home
12	End

Table 1-11: Key Field Values for the KEY Field for WIDGET_DRAW

Keyboard events are enabled using the KEYBOARD_EVENTS keyword to WIDGET_DRAW or the DRAW_KEYBOARD_EVENTS keyword to WIDGET_CONTROL. The DRAW_KEYBOARD_EVENTS keyword to WIDGET_INFO allows you to determine the current setting.

Scrolling Draw Widget Enhancements

In prior releases of IDL, the use of the APP_SCROLL keyword to WIDGET_DRAW caused the draw widget to be created with VIEWPORT_EVENTS = 1, RETAIN = 0, and EXPOSE_EVENTS = 1 regardless of the settings of these three keywords. As a result, if APP_SCROLL was set, you had to explicitly refresh the display when an expose event occurred.

In IDL version 5.6, for draw widgets that use Direct Graphics for their drawable areas, the settings for RETAIN and EXPOSE_EVENTS have been de-coupled from the setting of APP_SCROLL. This allows you to create a scrolling draw widget that can refresh an obscured part of the viewport from backing store. Since a refresh from backing store redraws only the newly-exposed portion of the viewport, a performance improvement may occur when backing store is used. (In prior releases, the viewport had to be redrawn in its entirety when the event handler received an expose event.)

Note

Draw widgets that use Object Graphics for their drawable areas are not affected by this change. If a draw widget uses Object Graphics and sets the APP_SCROLL keyword, IDL continues to behave as if RETAIN=0 and EXPOSE_EVENTS=1.

If a draw widget is created with APP_SCROLL set and RETAIN is set to 1 or 2, no expose events will be generated since the viewport will be refreshed from the backing store. If RETAIN is not set, the standard IDL default of RETAIN = 1 applies.

If you have existing code that sets APP_SCROLL = 1, RETAIN = 0 but does not set EXPOSE_EVENTS, the code will no longer refresh the viewport because the retain setting is maintained but expose events will not be generated. Code with these settings will need to be modified in one of the following ways:

- Remove the RETAIN = 0 setting. Which will cause IDL to use the default setting of RETAIN = 1, which makes use of the system backing store.
- Change RETAIN = 0 to RETAIN = 1. Which will explicitly specify that the system backing store be used.
- Change RETAIN = 0 to RETAIN = 2. Which will explicitly specify that IDL provide backing store.
- Leave RETAIN = 0 and set EXPOSE_EVENTS = 1. Which will restore the previous behavior, allowing you to explicitly handle expose events in an event-handling routine.

Note

The use of APP_SCROLL = 1 still causes viewport events to be generated regardless of the setting of the VIEWPORT_EVENTS keyword, since handling viewport events is fundamental to the use of a scrolling draw widget.

Label Widget Enhancements

The SUNKEN_FRAME keyword to WIDGET_LABEL has been added to create a three dimensional, bevelled border around the label widget. The resulting frame gives the label a sunken appearance, similar to what is often seen in application status bars.

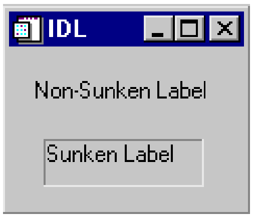


Figure 1-8: Sunken and Non-Sunken Labels

Enhancements to WIDGET_INFO

In addition to the enhancements to the WIDGET_INFO routine described above, the following new keywords have been added:

Keyword	Description
FONTNAME	Set this keyword to return a string containing the name of the font being used by the specified widget. The returned name can then be used when creating other widgets or with the SET_FONT keyword to the DEVICE procedure.
MAP	Set this keyword to return True (1) if the widget specified by *Widget_ID* is mapped (visible), or False (0) otherwise. Note that when a base widget is unmapped, all of its children are unmapped. If WIDGET_INFO reports that a particular widget is unmapped, it may be because a parent in the widget hierarchy has been unmapped.
SENSITIVE	Set this keyword to return True (1) if the widget specified by *Widget_ID* is sensitive (enabled), or False (0) otherwise. Note that when a base is made insensitive, all its children are made insensitive. If WIDGET_INFO reports that a particular widget is insensitive, it may be because a parent in the widget hierarchy has been made insensitive.
VISIBLE	Set this keyword to return True (1) if the widget specified by *Widget_ID* is visible, or False (0) otherwise. A widget is visible if: • It has been realized. • It and all of its ancestors are mapped.

Table 1-12: New WIDGET_INFO Keywords

New Personal Use Licensing

New with IDL 5.6, an IDL Personal Use License is associated with a designated user, not with a specific hardware device, so licenses can be easily moved. With a Personal Use License, RSI customers may use their IDL license in their office, lab, on a laptop computer, and even at home. The Personal Use License option will allow a single, named individual to install and license IDL on up to four (4) separate computer systems that share a common operating system, as long as the license is only used on a single computer at any one time. Available platforms include Windows, Linux and Mac OS X. RSI will continue to offer Node-Locked and Network Floating licenses for IDL.

New Support for Macintosh OS X

IDL 5.6 now supports the OS X operating system for Macintosh platforms. Important information about this release:

- Mac OS X is based on a UNIX operating system named Darwin. IDL 5.6 has been released upon Darwin. All standard UNIX features are available on the Macintosh OS X version of IDL, including multi-processor support with the IDL Thread Pool.
- IDL 5.6 for OS X does not run in any emulation mode. It is a fully native version of IDL. All GUI and graphical output are produced as X11 graphics.
- IDL 5.6 for OS X requires XFree86 version 4.2 (XDarwin 1.0.6.). This package supplements the Macintosh Quartz/Aqua window system with the ability to display X window graphics. This version has been included on your IDL 5.6 product CD.
- IDL 5.6 for OS X will look similar to the UNIX Motif interface. With the addition of the OroborOSX window manager, included on your IDL 5.6 product CD, you can enhance the Xwindow dressings to have the look and feel of the standard Aqua interface.
- Although the interface does not have the Aqua look and feel, the X11 interface does provide some unique advantages:
  - Mac OS X IDL is installed and administered identically to any other UNIX installation. It can be part of a multi-platform installation of IDL, and it can be located on a remote server.
  - As an X11 program, IDL can display its graphics on any remote X11 display on a network, including non-Apple systems.
  - IDL for Mac OS X supports Altivec to the same level as previous Macintosh IDL versions.
- IDL for Mac OS X supports node-locked, floating, and the new Personal Use style of licensing.

Documentation Enhancements

In addition to documentation for new and enhanced IDL features, the following enhancements to the IDL documentation set are included in the 5.6 release:

- New Image Processing in IDL Manual
- Revised and Enhanced External Development Guide
- Revised Graphical User Interface Documentation
- Version History in Reference Documentation
- New Online Help Systems

New *Image Processing in IDL* Manual

The new *Image Processing in IDL* manual introduces you to the full image processing power of IDL, describing how to display, manipulate, and extract information from images. Topics include:

- Working with color
- Texture mapping and Warping
- Applying transforms
- Applying filters
- Using morphological operators

This manual features both Direct Graphics and Object Graphics examples that will aid in developing IDL applications that require image processing.

Revised Graphical User Interface Documentation

"Part V: Creating Graphical User Interfaces in IDL" in the *Building IDL Applications* manual has been revised. Chapter 23, "Using the IDL GUIBuilder" has been updated with the changes made in this release and the "Widgets" chapter has been revised, rewritten, and broken into three chapters:

- Chapter 24, "Widgets"
- Chapter 25, "Creating Widget Applications"
- Chapter 26, "Widget Application Techniques"

Revised and Enhanced *External Development Guide*

The *External Development Guide* has been extensively updated to include features introduced in recent releases. Of particular note are the following chapters:

- Chapter 3, "Overview: COM and ActiveX in IDL"
- Chapter 4, "Using COM Objects in IDL"
- Chapter 5, "Using ActiveX Controls in IDL"
- Chapter 8, "CALL_EXTERNAL" (with emphasis on the AUTO_GLUE mechanism)

Version History in Reference Documentation

Documentation for routines and objects in the *IDL Reference Guide*, the *Scientific Data Formats* manual, and the *IDL DataMiner Guide*, and the Online Help now include a section describing when the routine was first included in IDL.

New Online Help Systems

Windows versions of IDL now use the Microsoft HTML Help viewer (based on Internet Explorer) to display help in Windows HTML Help format. In addition, the entire IDL documentation set is available in PDF format.

UNIX versions of IDL now use the Adobe Acrobat Reader software to display a set of hyperlinked Adobe Portable Document Format (PDF) files.

IDL no longer uses the Bristol HyperHelp viewer to provide online help for UNIX platforms. Instead, it now uses the free Adobe Acrobat Reader. Acrobat Reader version 3 or higher must be installed on your system, and the corresponding acroread command must be available from your Unix PATH environment variable. Acrobat reader is available from www.adobe.com as well as you IDL product CD-ROM. Acrobat Reader has many advantages:

- Output is publication quality, and printing is well supported.
- No cross platform issues: PDF is the same no matter where it is viewed.
- Tables and Figures look exactly as they appear in the published manuals.
- Supports hypertext style links, just as with HyperHelp, both within the current document and between documents.
- PDF files (the Acrobat format) are generated automatically from our books without the need for manual proofreading.

- The Acrobat Reader is freely available, and the use of the PDF format is widespread.
- In comparison to HyperHelp, Acrobat is an industry standard, and tools for producing PDF are available.

Significant time and effort were dedicated to identifying the best replacement for Bristol HyperHelp. While providing HTML online help using standard Web browsers was considered, the requirements of publication quality documentation led to the conclusion that the use of PDF is a superior fit for the IDL online help system. Benefits include more consistent formatting of tables and figures, quality printed output, and the lack of browser incompatibility issues.

Mechanisms for using the new help systems from within user-written applications are discussed in Chapter 20, “Providing Online Help For Your Application” in the *Building IDL Applications* manual.

Changes to ONLINE_HELP and the “?” Command

The ONLINE_HELP procedure and ? command have been modified in the following ways:

- On UNIX platforms, IDL uses the Adobe Acrobat Reader to display IDL’s online help files.
- On Windows platforms, IDL uses Microsoft HTML Help to display IDL’s online help files.
- On UNIX platforms, ONLINE_HELP can open an appropriate viewer and display files in Adobe Portable Document Format (PDF), or HTML format.
- On Windows platforms, ONLINE_HELP can open an appropriate viewer and display files in HTML Help, WinHelp, Adobe Portable Document Format (PDF), or HTML format.

Note

If you have created custom Help files using Bristol HyperHelp, you will no longer be able to access them using the ONLINE_HELP procedure.

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

New and Enhanced IDL Objects

This section describes the following:

- New IDL Object Classes
- New IDL Object Methods
- IDL Object Method Enhancements

New IDL Object Classes

The following table describes the new object classes in IDL 5.5 for Windows.

New Object Class	Description
IDLffXMLSAX	An IDLffXMLSAX object uses an XML SAX level 2 parser. The XML parser allows you to read an XML file and store arbitrary data from the file in IDL variables. The parser object's methods are *callbacks*. These methods are called automatically when the parser encounters different types of XML elements or attributes.

New IDL Object Methods

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

New Method	Description
IDLgrContour::GetLabelInfo	The IDLgrContour::GetLabelInfo procedure method retrieves information about the labels for the contour. The returned information is only valid until the next time the C_LABEL_INTERVAL or C_LABEL_OBJECTS property is modified via the IDLgrContour::SetProperty method, or the offsets are adjusted via IDLgrContour::AdjustLabelOffsets.

IDL Object Method Enhancements

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

IDLanROI::ComputeMask

Item	Description
RUN_LENGTH	Set this keyword to a non-zero value to return a run-length encoded representation of the mask, stored in a one-dimensional unsigned long array. When run-length encoded, each element with an even subscript contains the length of the run, and the following element contains the starting index of the run.

IDLanROIGroup::ComputeMask

Item	Description
RUN_LENGTH	Set this keyword to a non-zero value to return a run-length encoded representation of the mask, stored in a one-dimensional unsigned long array. When run-length encoded, each element with an even subscript contains the length of the run, and the following element contains the starting index of the run.

IDLffShape::GetProperty

Item	Description
N_RECORDS	Return the number of records in the dBASE table (`.dbf`) component of the shapefile. In a normal operating mode, this is accomplished by getting the number of entities. However, in DBF_ONLY mode, no entity file exits.

IDLffShape::Init

Item	Description
DBF_ONLY	If this keyword is set to a positive value, only the underlying dBASE table (`.dbf`) component of the shapefile is opened. All entity related files are left closed. Two values to this keyword are accepted: 1 - Open an existing `.dbf` file, > 1 - Create a new `.dbf` file The UPDATE keyword is required to open the `.dbf` file for updating.

IDLffShape::Open

Item	Description
DBF_ONLY	If this keyword is set to a positive value, only the underlying dBASE table (`.dbf`) component of the shapefile is opened. All entity related files are left closed. Two values to this keyword are accepted: 1 - Open an existing `.dbf` file, > 1 - Create a new `.dbf` file The UPDATE keyword is required to open the `.dbf` file for updating.

IDLgrAxis::Init

Item	Description
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 - The clipping planes specified via this keyword are applied in addition to the near and far clipping planes associated with the IDLgrView in which this object appears. Note - Clipping planes are applied in the data space of this object (prior to the application of any x, y, or z coordinate conversion). Note - To determine the maximum number of clipping planes supported by the device, use the MAX_NUM_CLIP_PLANES keyword of the GetDeviceInfo method for the IDLgrBuffer, IDLgrClipboard, IDLgrWindow, and IDLgrVRML objects.

IDLgrBuffer::GetDeviceInfo

Item	Description
MAX_NUM_CLIP_PLANES	Set this keyword to a named variable that upon return will contain an integer that specifies the maximum number of user-defined clipping planes supported by the device.

IDLgrBuffer::PickData

Item	Description
PICK_STATUS	Set this keyword to a named variable that will contain "hit" information for each pixel in the pick box. If the DIMENSIONS keyword is not set, the PICK_STATUS will be a scalar value exactly matching the *Result* of the method call. If the DIMENSIONS keyword is set, the PICK_STATUS variable will be an array matching the dimensions of the pick box. Each value in the PICK_STATUS array corresponds to a pixel in the pick box, and will be set to one of the following values: -1: if the pixel falls outside of the window's viewport. 0: if no graphic object is "hit" at that pixel location. 1: if a graphic object is "hit" at that pixel location.

IDLgrClipboard::GetDeviceInfo

Item	Description
MAX_NUM_CLIP_PLANES	Set this keyword to a named variable that upon return will contain an integer that specifies the maximum number of user-defined clipping planes supported by the device.

IDLgrContour::Init

Item	Description
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.

Item	Description
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_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 that the objects specified via this keyword are used as descriptors only. The actual objects drawn as labels are generated by IDL, and may be accessed via the IDLgrContour::GetLabelInfo method. The contour labels will have the same color as the corresponding 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.

Item	Description
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_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_USE_LABEL_ COLOR	Set this keyword to a vector of values 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.

Item	Description
C_USE_LABEL_ ORIENTATION	Set this keyword to a vector of values 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).
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 - The clipping planes specified via this keyword are applied in addition to the near and far clipping planes associated with the IDLgrView in which this object appears. Note - Clipping planes are applied in the data space of this object (prior to the application of any x, y, or z coordinate conversion). Note - To determine the maximum number of clipping planes supported by the device, use the MAX_NUM_CLIP_PLANES keyword of the GetDeviceInfo method for the IDLgrBuffer, IDLgrClipboard, IDLgrWindow, and IDLgrVRML objects.

Item	Description
DAYS_OF_WEEK	Set this keyword to a vector of 7 strings 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.
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, where: Axis is simply the value 2 to indicate that values along the Z axis are being formatted. (This 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.

Item	Description
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). Note - LABEL_FRMTDATA will not be included in the structure returned via the ALL keyword to the IDLgrContour::GetProperty method.
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').

Item	Description
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.
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).

IDLgrImage::Init

Item	Description
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 - The clipping planes specified via this keyword are applied in addition to the near and far clipping planes associated with the IDLgrView in which this object appears. Note - Clipping planes are applied in the data space of this object (prior to the application of any x, y, or z coordinate conversion). Note - To determine the maximum number of clipping planes supported by the device, use the MAX_NUM_CLIP_PLANES keyword of the GetDeviceInfo method for the IDLgrBuffer, IDLgrClipboard, IDLgrWindow, and IDLgrVRML objects.

IDLgrModel::Init

Item	Description
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 - The clipping planes specified via this keyword are applied in addition to the near and far clipping planes associated with the IDLgrView in which this object appears. Note - Clipping planes are applied in the data space of the objects this model contains (prior to the application of this model's transform). Note - To determine the maximum number of clipping planes supported by the device, use the MAX_NUM_CLIP_PLANES keyword of the GetDeviceInfo method for the IDLgrBuffer, IDLgrClipboard, IDLgrWindow, and IDLgrVRML objects.

IDLgrPlot::Init

Item	Description
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 - The clipping planes specified via this keyword are applied in addition to the near and far clipping planes associated with the IDLgrView in which this object appears. Note - Clipping planes are applied in the data space of this object (prior to the application of any x, y, or z coordinate conversion). Note - To determine the maximum number of clipping planes supported by the device, use the MAX_NUM_CLIP_PLANES keyword of the GetDeviceInfo method for the IDLgrBuffer, IDLgrClipboard, IDLgrWindow, and IDLgrVRML objects.

IDLgrPolygon::Init

Item	Description
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 - The clipping planes specified via this keyword are applied in addition to the near and far clipping planes associated with the IDLgrView in which this object appears. Note - Clipping planes are applied in the data space of this object (prior to the application of any x, y, or z coordinate conversion). Note - To determine the maximum number of clipping planes supported by the device, use the MAX_NUM_CLIP_PLANES keyword of the GetDeviceInfo method for the IDLgrBuffer, IDLgrClipboard, IDLgrWindow, and IDLgrVRML objects.

IDLgrPolyline::Init

Item	Description
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 - The clipping planes specified via this keyword are applied in addition to the near and far clipping planes associated with the IDLgrView in which this object appears. Note - Clipping planes are applied in the data space of this object (prior to the application of any x, y, or z coordinate conversion). Note - To determine the maximum number of clipping planes supported by the device, use the MAX_NUM_CLIP_PLANES keyword of the GetDeviceInfo method for the IDLgrBuffer, IDLgrClipboard, IDLgrWindow, and IDLgrVRML objects.
LABEL_NOGAPS	Set this keyword to a vector of values indicating whether gaps should be computed for the corresponding label. A zero value indicates that a gap will be computed for the labels; a non-zero value indicates that no gap will be computed for the label. If the number of labels exceeds the number of elements in this vector, the LABEL_NOGAPS values will be repeated cyclically. By default, gaps are computed for all labels (so that the polyline does not pass through the label).

Item	Description
LABEL_OFFSETS	Set this keyword to a scalar or vector of floating point offsets, [t0, t1, …], that indicate the parametric offsets along the length of each polyline (specified via the LABEL_POLYLINES keyword) at which each label (as specified via the LABEL_OBJECTS keyword) would be positioned. If LABEL_OFFSETS is set to a scalar less than zero, then the offsets will be automatically computed to be evenly distributed along the length of the polyline. If a scalar value greater than or equal to zero is provided, it is used for all labels. If a vector is provided, the number of offsets must match the number of labels provided via LABEL_OBJECTS. By default, this keyword is set to the scalar, -1, indicating that the label offsets will be automatically computed.

Item	Description
LABEL_OBJECTS	Set this keyword to an object reference (or vector of object references) to specify the labels to be drawn along the polyline path(s). 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 label. If a vector of objects is used, any IDLgrText objects should have only a single string; each object will correspond to a label. If one or more IDLgrText objects are provided, the LOCATION property of the provided text object(s) may be overwritten; position is determined according to the values provided via the LABEL_OFFSETS keyword. The labels will have the same color as the corresponding polyline (see the COLOR keyword) unless the USE_LABEL_COLOR keyword is specified. The orientation of the label objects USE_LABEL_ORIENTATION keyword is specified. The horizontal and vertical alignment for any text labels will each default to 0.5 (i.e., centered) unless the USE_TEXT_ALIGNMENTS keyword is specified. Note - The objects provided via this keyword will not be destroyed automatically when this IDLgrPolyline is destroyed.

Item	Description
LABEL_POLYLINES	Set this keyword to a scalar or a vector of polyline indices, [P0, P1, …], that indicate which polylines are to be labeled. Pi corresponds to the ith polyline specified via the POLYLINES keyword. This keyword is intended to be used in conjunction with the LABEL_OBJECTS keyword. If a scalar is provided, all labels will be drawn along the single indicated polyline. If a vector is provided, the number of polyline indices must match the number of labels provided via LABEL_OBJECTS. By default, this keyword is set to the scalar, 0, indicating that only the first polyline will be labeled. Note - If a given polyline has more than one label, then the corresponding polyline index may appear more than once in the LABEL_POLYLINES vector.
LABEL_USE_VERTEX_COLOR	Set this keyword to a non-zero value to indicate that labels should be colored according to the vertex coloring (if the VERT_COLORS keyword is set). By default, this value is zero, indicating that the label will be drawn using the color specified via the COLOR property of the polyline object (unless the USE_LABEL_COLOR keyword is set).
USE_LABEL_COLOR	Set this keyword to a vector of values to indicate whether the COLOR property value for the corresponding label object is to be used to draw that label. If the number of labels exceeds the number of elements in this vector, the USE_LABEL_COLOR values will be repeated cyclically. By default, this value is zero, indicating that the COLOR property of each label object will be ignored, and the COLOR property for the polyline object will be used instead.

Item	Description
USE_LABEL_ORIENTATION	Set this keyword to a vector of values to indicate whether the orientation of the corresponding label object is to be used to draw that label. For IDLgrText objects, this refers to the BASELINE and UPDIR property values. For IDLgrSymbol objects, this refers to the default (un-rotated) orientation of the symbol. If the number of labels exceeds the number of elements in this vector, the USE_LABEL_ORIENTATION values will be repeated cyclically. By default, USE_LABEL_ORIENTATION is zero, indicating that the orientation will be automatically computed so that the baseline is parallel to the polyline, and the updir is perpendicular to the polyline.
USE_TEXT_ALIGNMENTS	Set this keyword to indicate that, for any IDLgrText labels (as specified via the LABEL_OBJECTS keyword), the ALIGNMENT and VERTICAL_ALIGNMENT property values for the given IDLgrText object(s) are to be used to draw those labels. By default, this value is zero, indicating that the ALIGNMENT and VERTICAL_ALIGNMENT properties of the IDLgrText object(s) will be overwritten with default values (0.5 for each, indicating centered labels).

IDLgrROI::Init

Item	Description
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 - The clipping planes specified via this keyword are applied in addition to the near and far clipping planes associated with the IDLgrView in which this object appears. Note - Clipping planes are applied in the data space of this object (prior to the application of any x, y, or z coordinate conversion). Note - To determine the maximum number of clipping planes supported by the device, use the MAX_NUM_CLIP_PLANES keyword of the GetDeviceInfo method for the IDLgrBuffer, IDLgrClipboard, IDLgrWindow, and IDLgrVRML objects.

IDLgrROIGroup::Init

Item	Description
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 - The clipping planes specified via this keyword are applied in addition to the near and far clipping planes associated with the IDLgrView in which this object appears. Note - Clipping planes are applied in the data space of this object (prior to the application of any x, y, or z coordinate conversion). Note - To determine the maximum number of clipping planes supported by the device, use the MAX_NUM_CLIP_PLANES keyword of the GetDeviceInfo method for the IDLgrBuffer, IDLgrClipboard, IDLgrWindow, and IDLgrVRML objects.

IDLgrSurface::Init

Item	Description
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 - The clipping planes specified via this keyword are applied in addition to the near and far clipping planes associated with the IDLgrView in which this object appears. Note - Clipping planes are applied in the data space of this object (prior to the application of any x, y, or z coordinate conversion). Note - To determine the maximum number of clipping planes supported by the device, use the MAX_NUM_CLIP_PLANES keyword of the GetDeviceInfo method for the IDLgrBuffer, IDLgrClipboard, IDLgrWindow, and IDLgrVRML objects.

IDLgrSymbol::Init

Item	Description
Data	This argument can contain either an integer value from the list shown below, or an object reference to either an IDLgrModel object or atomic graphic object. Use 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 If an instance of the IDLgrModel object class or an atomic graphic object is used, the object tree is used as the symbol.

IDLgrTessellator::AddPolygon

Item	Description
AUXDATA	Set this keyword to an array of auxiliary per-vertex data. This array must have dimensions [*m*,*n*] where m is the number of auxiliary data items per vertex and n is the number of vertices specified in the X, Y, and Z arguments. If you specify AUXDATA in any invocation of the AddPolygon method, you must specify it on all invocations of the method for the polygons to be tessellated together with the Tessellate method. Further, the value of m in the dimensions must be the same for all polygons. That is, all polygons must have the same number of auxiliary data items for each vertex.

IDLgrTessellator::Tessellate

Item	Description
AUXDATA	Set this keyword to a named variable that receives the auxiliary data associated with each vertex returned in the Vertices argument. The data is an [*m*, *n*] array where m is the number of per-vertex auxiliary data items specified in the call(s) to the AddPolygon method, and n is the number of vertices returned in the Vertices argument. The type of the returned auxiliary data is the same as the type of the data supplied with the AUXDATA keyword in the last call to AddPolygon.

IDLgrText::Init

Item	Description
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 - The clipping planes specified via this keyword are applied in addition to the near and far clipping planes associated with the IDLgrView in which this object appears. Note - Clipping planes are applied in the data space of this object (prior to the application of any x, y, or z coordinate conversion). Note - To determine the maximum number of clipping planes supported by the device, use the MAX_NUM_CLIP_PLANES keyword of the GetDeviceInfo method for the IDLgrBuffer, IDLgrClipboard, IDLgrWindow, and IDLgrVRML objects.

IDLgrVolume::Init

Item	Description
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 - The clipping planes specified via this keyword are applied in addition to the near and far clipping planes associated with the IDLgrView in which this object appears. Note - Clipping planes are applied in the data space of this object (prior to the application of any x, y, or z coordinate conversion). Note - To determine the maximum number of clipping planes supported by the device, use the MAX_NUM_CLIP_PLANES keyword of the GetDeviceInfo method for the IDLgrBuffer, IDLgrClipboard, IDLgrWindow, and IDLgrVRML objects. 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).

IDLgrVRML::GetDeviceInfo

Item	Description
MAX_NUM_CLIP_PLANES	Set this keyword to a named variable that upon return will contain an integer that specifies the maximum number of user-defined clipping planes supported by the device.

IDLgrWindow::GetDeviceInfo

Item	Description
MAX_NUM_CLIP_PLANES	Set this keyword to a named variable that upon return will contain an integer that specifies the maximum number of user-defined clipping planes supported by the device.

IDLgrWindow::PickData

Item	Description
PICK_STATUS	Set this keyword to a named variable that will contain “hit” information for each pixel in the pick box. If the DIMENSIONS keyword is not set, the PICK_STATUS will be a scalar value exactly matching the *Result* of the method call. If the DIMENSIONS keyword is set, the PICK_STATUS variable will be an array matching the dimensions of the pick box. Each value in the PICK_STATUS array corresponds to a pixel in the pick box, and will be set to one of the following values: -1: if the pixel falls outside of the window’s viewport. 0: if no graphic object is “hit” at that pixel location. 1: if a graphic object is “hit” at that pixel location.

IDLgrWindow::SetCurrentCursor

Item	Description
CursorName	A string that specifies which built-in cursor to use. This argument is ignored if any keywords to this routine are set. This string can either be a name provided to the REGISTER_CURSOR routine or one of the following: • ARROW • CROSSHAIR • ICON • IBEAM • MOVE • ORIGINAL • SIZE_NE • SIZE_NW • SIZE_SE • SIZE_SW • SIZE_NS • SIZE_EW • UP_ARROW

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
COPY_LUN	The COPY_LUN procedure copies data between two open files. It is useful in situations where it is necessary to transfer a known amount of data from one file to another without the requirement of having the data available in an IDL variable.
DIAG_MATRIX	The DIAG_MATRIX procedure constructs a diagonal matrix from an input vector, or if given a matrix, then DIAG_MATRIX will extract a diagonal vector.
FILE_COPY	The FILE_COPY procedure copies files, or directories of files, to a new location. The copies retain the protection settings of the original files, and belong to the user that performed the copy.
FILE_LINES	The FILE_LINES function returns the number of lines of text contained within the specified file or files. If an array of file names is specified as the input parameter, the return value is an array with the same number of elements as the input array, with each element containing the number of lines in the corresponding file.

Table 1-13: New Routines in IDL 5.6

New Routine	Description
FILE_LINK	The FILE_LINK procedure creates UNIX file links, both regular (hard) and symbolic. FILE_LINK is available only under UNIX.
FILE_MOVE	The FILE_MOVE procedure renames files and directories. The moved files retain their protection and ownership attributes.
FILE_READLINK	The FILE_READLINK function returns the path pointed to by UNIX symbolic links.
FILE_SAME	The FILE_SAME function is used to determine if two different file names refer to the same underlying file. FILE_SAME returns True (1) if they are, or False (0) otherwise. If either or both of the input arguments are arrays of file names, the result is an array, following the same rules as standard IDL operators.
H5_BROWSER	The H5_BROWSER function presents a graphical user interface for viewing and reading HDF5 files.
H5_CLOSE	The H5_CLOSE procedure flushes all data to disk, closes file identifiers, and cleans up memory. This routine closes IDL's link to its HDF5 libraries. This procedure is used automatically by IDL when RESET_SESSION is issued, but it may also be called if the user desires to free all HDF5 resources.
H5_GET_LIBVERSION	The H5_GET_LIBVERSION function returns the current version of the HDF5 library used by IDL.
H5_OPEN	The H5_OPEN procedure initializes IDL's HDF5 library. This procedure is issued automatically by IDL when one of IDL's HDF5 routines is used.

Table 1-13: New Routines in IDL 5.6 (Continued)

New Routine	Description
H5_PARSE	The H5_PARSE function recursively descends through an HDF5 file or group and creates an IDL structure containing object information and data.
H5A_CLOSE	The H5A_CLOSE procedure closes the specified attribute and releases resources used by it. After this routine is used, the attribute's identifier is no longer available until the H5A_OPEN routines are used again to specify that attribute. Further use of the attribute identifier is illegal.
H5A_GET_NAME	The H5A_GET_NAME function retrieves an attribute name given the attribute identifier number.
H5A_GET_NUM_ATTRS	The H5A_GET_NUM_ATTRS function returns the number of attributes attached to a group, dataset, or a named datatype.
H5A_GET_SPACE	The H5A_GET_SPACE function returns the identifier number of a copy of the dataspace for an attribute.
H5A_GET_TYPE	The H5A_GET_TYPE function returns the identifier number of a copy of the datatype for an attribute.
H5A_OPEN_IDX	The H5A_OPEN_IDX function opens an existing attribute by the index of that attribute within an HDF5.
H5A_OPEN_NAME	The H5A_OPEN_NAME function opens an existing attribute by the name of that attribute within an HDF5 file.
H5A_READ	The H5A_READ function reads the data within an attribute, converting from the HDF5 file datatype into the HDF5 memory datatype, and finally into the corresponding IDL datatype.

Table 1-13: New Routines in IDL 5.6 (Continued)

New Routine	Description
H5D_CLOSE	The H5D_CLOSE procedure closes the specified dataset and releases its used resources.
H5D_GET_SPACE	The H5D_GET_SPACE function returns an identifier number for a copy of the dataspace for a dataset.
H5D_GET_STORAGE_SIZE	The H5D_GET_STORAGE_SIZE function returns the amount of storage in bytes required for a dataset. For chunked datasets this is the number of allocated chunks times the chunk size.
H5D_GET_TYPE	The H5D_GET_TYPE function returns an identifier number for a copy of the datatype for a dataset.
H5D_OPEN	The H5D_OPEN function opens an existing dataset within an HDF5 file.
H5D_READ	The H5D_READ function reads the data within a dataset, converting from the HDF5 file datatype into the HDF5 memory datatype, and finally into the corresponding IDL datatype.
H5F_CLOSE	The H5F_CLOSE procedure closes the specified file and releases resources used by it.
H5F_IS_HDF5	The H5F_IS_HDF5 function determines if a file is in the HDF5 format.
H5F_OPEN	The H5F_OPEN function opens an existing HDF5 file.
H5G_CLOSE	The H5G_CLOSE procedure closes the specified group and releases resources used by it.
H5G_GET_COMMENT	The H5G_GET_COMMENT function retrieves a comment string from a specified object.

Table 1-13: New Routines in IDL 5.6 (Continued)

New Routine	Description
H5G_GET_LINKVAL	The H5G_GET_LINKVAL function returns the name of the object pointed to by a symbolic link.
H5G_GET_MEMBER_NAME	The H5G_GET_MEMBER_NAME function retrieves the name of an object within a group, by its zero-based index.
H5G_GET_NMEMBERS	The H5G_GET_NMEMBERS function returns the number of objects within a group.
H5G_GET_OBJINFO	The H5G_GET_OBJINFO function retrieves information from a specified object.
H5G_OPEN	The H5G_OPEN function opens an existing group within an HDF5 file.
H5I_GET_TYPE	The H5I_GET_TYPE function returns the object's type.
H5R_DEREFERENCE	The H5R_DEREFERENCE function opens a reference and returns the object identifier.
H5R_GET_OBJECT_TYPE	The H5R_GET_OBJECT_TYPE function returns the type of object that an object reference points to.
H5S_CLOSE	The H5S_CLOSE procedure releases and terminates access to a dataspace. After this routine is used, the dataspace's identifier is no longer available.
H5S_COPY	The H5S_COPY function copies an existing dataspace.
H5S_CREATE_SIMPLE	The H5S_CREATE_SIMPLE function creates a simple dataspace.
H5S_GET_SELECT_BOUNDS	The H5S_GET_SELECT_BOUNDS function retrieves the coordinates of the bounding box containing the current dataspace selection.

Table 1-13: New Routines in IDL 5.6 (Continued)

New Routine	Description
H5S_GET_SELECT_ELEM_NPOINTS	The H5S_GET_SELECT_ELEM_NPOINTS function determines the number of element points in the current dataspace selection.
H5S_GET_SELECT_ELEM_POINTLIST	The H5S_SELECT_ELEM_POINTLIST function returns a list of the element points in the current dataspace selection.
H5S_GET_SELECT_HYPER_BLOCKLIST	The H5S_GET_SELECT_HYPER_ BLOCKLIST function returns a list of the hyperslab blocks in the current dataspace selection.
H5S_GET_SELECT_HYPER_NBLOCKS	The H5S_GET_SELECT_HYPER_NBLOCKS function determines the number of hyperslab blocks in the current dataspace selection.
H5S_GET_SELECT_NPOINTS	The H5S_GET_SELECT_NPOINTS function determines the number of elements in a dataspace selection.
H5S_GET_SIMPLE_EXTENT_DIMS	The H5S_GET_SIMPLE_EXTENT_DIMS function returns the dimension sizes for a dataspace.
H5S_GET_SIMPLE_EXTENT_NDIMS	The H5S_GET_SIMPLE_EXTENT_NDIMS function determines the number of dimensions (or rank) of a dataspace.
H5S_GET_SIMPLE_EXTENT_NPOINTS	The H5S_GET_SIMPLE_EXTENT_NPOINTS function determines the number of elements in a dataspace.
H5S_GET_SIMPLE_EXTENT_TYPE	The H5S_GET_SIMPLE_EXTENT_TYPE function returns the current class of a dataspace.
H5S_IS_SIMPLE	The H5S_IS_SIMPLE function determines whether a dataspace is a simple dataspace.

Table 1-13: New Routines in IDL 5.6 (Continued)

New Routine	Description
H5S_OFFSET_SIMPLE	The H5S_OFFSET_SIMPLE procedure sets the selection offset for a simple dataspace. The offset allows the same shaped selection to be moved to different locations within the dataspace.
H5S_SELECT_ALL	The H5S_SELECT_ALL procedure selects the entire extent of a dataspace.
H5S_SELECT_ELEMENTS	The H5S_SELECT_ELEMENTS procedure selects array elements to be included in the selection for a dataspace.
H5S_SELECT_HYPERSLAB	The H5S_SELECT_HYPERSLAB procedure selects a hyperslab region to be included in the selection for a dataspace.
H5S_SELECT_NONE	The H5S_SELECT_NONE procedure resets the dataspace selection region to include no elements.
H5S_SELECT_VALID	The H5S_SELECT_VALID function verifies that the selection is within the extent of a dataspace.
H5T_CLOSE	The H5T_CLOSE procedure releases the specified datatype’s identifier and releases resources used by it.
H5T_COMMITTED	The H5T_COMMITTED function determines whether a datatype is a named datatype or a transient type.
H5T_COPY	The H5T_COPY function copies an existing datatype. The returned type is transient and unlocked.
H5T_EQUAL	The H5T_EQUAL function determines whether two datatype identifiers refer to the same datatype.

Table 1-13: New Routines in IDL 5.6 (Continued)

New Routine	Description
H5T_GET_ARRAY_DIMS	The H5T_GET_ARRAY_DIMS function returns the dimension sizes for an array datatype object.
H5T_GET_ARRAY_NDIMS	The H5T_GET_ARRAY_NDIMS function determines the number of dimensions (or rank) of an array datatype object.
H5T_GET_CLASS	The H5T_GET_CLASS function returns the datatype’s class.
H5T_GET_CSET	The H5T_GET_CSET function returns the character set type of a string datatype.
H5T_GET_EBIAS	The H5T_GET_EBIAS function returns the exponent bias of a floating-point type.
H5T_GET_FIELDS	The H5T_GET_FIELDS function retrieves information about the positions and sizes of bit fields within a floating-point datatype.
H5T_GET_INPAD	The H5T_GET_INPAD function returns the padding method for unused internal bits within a floating-point datatype.
H5T_GET_MEMBER_CLASS	The H5T_GET_MEMBER_CLASS function returns the datatype class of a compound datatype member.
H5T_GET_MEMBER_NAME	The H5T_GET_MEMBER_NAME function returns the datatype name of a compound datatype member.
H5T_GET_MEMBER_OFFSET	The H5T_GET_MEMBER_OFFSET function returns the byte offset of a field within a compound datatype.
H5T_GET_MEMBER_TYPE	The H5T_GET_MEMBER_TYPE function returns the datatype identifier for a specified member within a compound datatype.

Table 1-13: New Routines in IDL 5.6 (Continued)

New Routine	Description
H5T_GET_NMEMBERS	The H5T_GET_NMEMBERS function returns the number of fields in a compound datatype.
H5T_GET_NORM	The H5T_GET_NORM function returns the mantissa normalization of a floating-point datatype.
H5T_GET_OFFSET	The H5T_GET_OFFSET function returns the bit offset of the first significant bit in an atomic datatype. The offset is the number of bits of padding that follows the significant bits (for big endian) or precedes the significant bits (for little endian).
H5T_GET_ORDER	The H5T_GET_ORDER function returns the byte order of an atomic datatype.
H5T_GET_PAD	The H5T_GET_PAD function returns the padding method of the least significant bit (lsb) and most significant bit (msb) of an atomic datatype.
H5T_GET_PRECISION	The H5T_GET_PRECISION function returns the precision in bits of an atomic datatype. The precision is the number of significant bits which, unless padded, is 8 times larger than the byte size from H5T_GET_SIZE.
H5T_GET_SIGN	The H5T_GET_SIGN function returns the sign type for an integer datatype.
H5T_GET_SIZE	The H5T_GET_SIZE function returns the size of a datatype in bytes.
H5T_GET_STRPAD	The H5T_GET_STRPAD function returns the padding method for a string datatype.
H5T_GET_SUPER	The H5T_GET_SUPER function returns the base datatype from which a datatype is derived.
H5T_IDLTYPE	The H5T_IDLTYPE function returns the IDL type code corresponding to a datatype.

Table 1-13: New Routines in IDL 5.6 (Continued)

New Routine	Description
H5T_MEMTYPE	The H5T_MEMTYPE function returns the native memory datatype corresponding to a file datatype.
H5T_OPEN	The H5T_OPEN function opens a named datatype.
LA_CHOLDC	The LA_CHOLDC procedure computes the Cholesky factorization of an *n*-by-*n* symmetric (or Hermitian) positive-definite array as: • If A is real: $A = U^T U$ *or* $A = L L^T$ • If A is complex: $A = U^H U$ *or* $A = L L^H$ where *U* and *L* are upper and lower triangular arrays. The *T* represents the transpose while *H* represents the Hermitian, or transpose complex conjugate.
LA_CHOLMPROVE	The LA_CHOLMPROVE function uses Cholesky factorization to improve the solution to a system of linear equations, $AX = B$ (where *A* is symmetric or Hermitian), and provides optional error bounds and backward error estimates. The result is an *n*-element vector whose type is identical to *A*. The LA_CHOLMPROVE function may also be used to improve the solutions for multiple systems of linear equations, with each column of *B* representing a different set of equations. In this case, the result is a *k*-by-*n* array where each of the *k* columns represents the improved solution vector for that set of equations.

Table 1-13: New Routines in IDL 5.6 (Continued)

New Routine	Description
LA_CHOLSOL	The LA_CHOLSOL function is used in conjunction with the LA_CHOLDC procedure to solve a set of *n* linear equations in *n* unknowns, $AX = B$, where *A* must be a symmetric (or Hermitian) positive-definite array. The parameter *A* is input not as the original array, but as its Cholesky decomposition, created by the routine LA_CHOLDC. The result is an *n*-element vector whose type is identical to *A*. The LA_CHOLSOL function may also be used to solve for multiple systems of linear equations, with each column of *B* representing a different set of equations. In this case, the result is a *k*-by-*n* array where each of the *k* columns represents the solution vector for that set of equations.
LA_DETERM	The LA_DETERM function uses LU decomposition to compute the determinant of a square array. The result is a scalar of the same type as the input array. This routine is written in the IDL language. Its source code can be found in the file `la_determ.pro` in the `lib` subdirectory of the IDL distribution.

Table 1-13: New Routines in IDL 5.6 (Continued)

New Routine	Description
LA_EIGENPROBLEM	The LA_EIGENPROBLEM function uses the QR algorithm to compute all eigenvalues λ and eigenvectors $v \neq 0$ of an n-by-n real nonsymmetric or complex non-Hermitian array A, for the eigenproblem $Av = \lambda v$. The routine can also compute the left eigenvectors $u \neq 0$, which satisfy $u^H A = \lambda u^H$. LA_EIGENPROBLEM may also be used for the generalized eigenproblem: • $Av = \lambda Bv$ *and* $u^H A = \lambda u^H B$ where A and B are square arrays, v are the right eigenvectors, and u are the left eigenvectors. The result is a complex n-element vector containing the eigenvalues.
LA_EIGENQL	The LA_EIGENQL function computes selected eigenvalues λ and eigenvectors $z \neq 0$ of an n-by-n real symmetric or complex Hermitian array A, for the eigenproblem $Az = \lambda z$. LA_EIGENQL may also be used for the generalized symmetric eigenproblems: • $Az = \lambda Bz$ or $ABz = \lambda z$ or $BAz = \lambda z$ where A and B are symmetric (or Hermitian) and B is positive definite. In both cases, the result is a real vector containing the eigenvalues in ascending order.

Table 1-13: New Routines in IDL 5.6 (Continued)

New Routine	Description
LA_EIGENVEC	The LA_EIGENVEC function uses the QR algorithm to compute all or some of the eigenvectors $v \neq 0$ of an n-by-n real nonsymmetric or complex non-Hermitian array A, for the eigenproblem $Av = \lambda v$. The routine can also compute the left eigenvectors $u \neq 0$, which satisfy $u^H A = \lambda u^H$. The result is a complex array containing the eigenvectors as a set of row vectors.
LA_ELMHES	The LA_ELMHES function reduces a real nonsymmetric or complex non-Hermitian array to upper Hessenberg form H. If the array is real then the decomposition is $A = Q\ H\ Q^T$, where Q is orthogonal. If the array is complex Hermitian then the decomposition is $A = Q\ H\ Q^H$, where Q is unitary. The ^{T} represents the transpose while superscript H represents the Hermitian, or transpose complex conjugate. The result is an array of the same type as A containing the upper Hessenberg form.
LA_GM_LINEAR_MODEL	The LA_GM_LINEAR_MODEL function is used to solve a general Gauss-Markov linear model problem: • minimize$_x$ $\|\|y\|\|^2$ with constraint $d = Ax + By$ where A is an m-column by n-row array, B is a p-column by n-row array, and d is an n-element input vector with $m \leq n \leq m+p$. The result, x, is an m-element vector whose type is identical to A.

Table 1-13: New Routines in IDL 5.6 (Continued)

New Routine	Description
LA_HQR	The LA_HQR function uses the multishift QR algorithm to compute all eigenvalues of an *n*-by-*n* upper Hessenberg array. The LA_ELMHES routine can be used to reduce a real or complex array to upper Hessenberg form suitable for input to this procedure. LA_HQR may also be used to compute the matrices *T* and *QZ* from the Schur decomposition $A = (QZ)\ T\ (QZ)^H$. The result is an *n*-element complex vector.
LA_INVERT	The LA_INVERT function uses LU decomposition to compute the inverse of a square array. The result is an array of the same dimensions as the input array.
LA_LEAST_SQUARE_EQUALITY	The LA_LEAST_SQUARE_EQUALITY function is used to solve the linear least-squares problem: $\text{Minimize}_x \ \|\|Ax - c\|\|^2$ with constraint $Bx = d$ where *A* is an *n*-column by *m*-row array, *B* is an *n*-column by *p*-row array, *c* is an m-element input vector, and *d* is a *p*-element input vector with $p \le n \le m+p$. The result, *x*, is an *n*-element vector. If *B* has full row rank p and the array $\begin{pmatrix} A \\ B \end{pmatrix}$ has full column rank *n*, then a unique solution exists.

Table 1-13: New Routines in IDL 5.6 (Continued)

New Routine	Description
LA_LEAST_SQUARES	The LA_LEAST_SQUARES function is used to solve the linear least-squares problem: $\text{Minimize}_x \|\|Ax - b\|\|^2$ where A is a (possibly rank-deficient) n-column by m-row array, b is an m-element input vector, and x is the n-element solution vector.
LA_LINEAR_EQUATION	The LA_LINEAR_EQUATION function uses LU decomposition to solve a system of linear equations, $AX = B$, and provides optional error bounds and backward error estimates. The result is an n-element vector whose type is identical to A. The LA_LINEAR_EQUATION function may also be used to solve for multiple systems of linear equations, with each column of B representing a different set of equations. In this case, the result is a k-by-n array where each of the k columns represents the improved solution vector for that set of equations.
LA_LUDC	The LA_LUDC procedure computes the LU decomposition of an n-column by m-row array as: $A = P\ L\ U$ where P is a permutation matrix, L is lower trapezoidal with unit diagonal elements (lower triangular if $n = m$), and U is upper trapezoidal (upper triangular if $n = m$).

Table 1-13: New Routines in IDL 5.6 (Continued)

New Routine	Description
LA_LUMPROVE	The LA_LUMPROVE function uses LU decomposition to improve the solution to a system of linear equations, *AX* = *B*, and provides optional error bounds and backward error estimates. The result is an *n*-element vector. The LA_LUMPROVE function may also be used to improve the solutions for multiple systems of linear equations, with each column of *B* representing a different set of equations. In this case, the result is a *k*-by-*n* array where each of the *k* columns represents the improved solution vector for that set of equations.
LA_LUSOL	The LA_LUSOL function is used in conjunction with the LA_LUDC procedure to solve a set of *n* linear equations in *n* unknowns, *AX* = *B*. The parameter *A* is not the original array, but its LU decomposition, created by the routine LA_LUDC. The result is an *n*-element vector. The LA_LUSOL function may also be used to solve for multiple systems of linear equations, with each column of *B* representing a different set of equations. In this case, the result is a *k*-by-*n* array where each of the *k* columns represents the solution vector for that set of equations.

Table 1-13: New Routines in IDL 5.6 (Continued)

New Routine	Description
LA_SVD	The LA_SVD procedure computes the singular value decomposition (SVD) of an *n*-columns by *m*-row array as the product of orthogonal and diagonal arrays: • *A* is real: $A = U\,S\,V^T$ • *A* is complex: $A = U\,S\,V^H$ where *U* is an orthogonal array containing the left singular vectors, *S* is a diagonal array containing the singular values, and *V* is an orthogonal array containing the right singular vectors. The superscript *T* represents the transpose while the superscript *H* represents the Hermitian, or transpose complex conjugate. If $n < m$ then *U* has dimensions (*n* x *m*), *S* has dimensions (*n* x *n*), and V^H has dimensions (*n* x *n*). If $n \geq m$ then *U* has dimensions (*m* x *m*), *S* has dimensions (*m* x *m*), and V^H has dimensions (*n* x *m*).
LA_TRIDC	The LA_TRIDC procedure computes the LU decomposition of a tridiagonal (*n* x *n*) array as $Array = L\,U$, where *L* is a product of permutation and unit lower bidiagonal arrays, and *U* is upper triangular with nonzero elements only in the main diagonal and the first two superdiagonals.

Table 1-13: New Routines in IDL 5.6 (Continued)

New Routine	Description
LA_TRIMPROVE	The LA_TRIMPROVE function improves the solution to a system of linear equations with a tridiagonal array, $AX = B$, and provides optional error bounds and backward error estimates. The result is an *n*-element vector. The LA_TRIMPROVE function may also be used to improve the solutions for multiple systems of linear equations, with each column of *B* representing a different set of equations. In this case, the result is a *k*-by-*n* array where each of the *k* columns represents the improved solution vector for that set of equations.
LA_TRIQL	The LA_TRIQL procedure uses the QL and QR variants of the implicitly-shifted QR algorithm to compute the eigenvalues and eigenvectors of a symmetric tridiagonal array. The LA_TRIRED routine can be used to reduce a real symmetric (or complex Hermitian) array to tridiagonal form suitable for input to this procedure.
LA_TRIRED	The LA_TRIRED procedure reduces a real symmetric or complex Hermitian array to real tridiagonal form *T*. If the array is real symmetric then the decomposition is $A = Q\ T\ Q^T$, where *Q* is orthogonal. If the array is complex Hermitian then the decomposition is $A = Q\ T\ Q^H$, where *Q* is unitary. The superscript *T* represents the transpose while superscript *H* represents the Hermitian, or transpose complex conjugate.

Table 1-13: New Routines in IDL 5.6 (Continued)

New Routine	Description
LA_TRISOL	The LA_TRISOL function is used in conjunction with the LA_TRIDC procedure to solve a set of *n* linear equations in n unknowns, *AX* = *B*, where *A* is a tridiagonal array. The parameter *A* is input not as the original array, but as its LU decomposition, created by the routine LA_TRIDC. The result is an *n*-element vector. The LA_TRISOL function may also be used to solve for multiple systems of linear equations, with each column of *B* representing a different set of equations. In this case, the result is a *k*-by-*n* array where each of the *k* columns represents the solution vector for that set of equations.
MAP_PROJ_FORWARD	The MAP_PROJ_FORWARD function transforms map coordinates from longitude/latitude to (X, Y) Cartesian coordinates, using either the !MAP system variable or a supplied map projection variable.
MAP_PROJ_INIT	The MAP_PROJ_INIT function initializes a mapping projection, using either the IDL or General Cartographic Transformation Package (GCTP) map projections. The result is a !MAP structure containing the map parameters, which can be used as input to the map transformation functions MAP_PROJ_FORWARD and MAP_PROJ_INVERSE.
MAP_PROJ_INVERSE	The MAP_PROJ_INVERSE function transforms map coordinates from (X, Y) Cartesian coordinates to longitude/latitude, using either the !MAP system variable or a supplied map projection variable.

Table 1-13: New Routines in IDL 5.6 (Continued)

New Routine	Description
MATRIX_POWER	The MATRIX_POWER function computes the product of a matrix with itself. For example, the fifth power of array *A* is *A* # *A* # *A* # *A* # *A*. Negative powers are computed using the matrix inverse of the positive power.
PRODUCT	The PRODUCT function returns the product of elements within an array. The product of the array elements can also be computed over a given dimension.
REGISTER_CURSOR	The REGISTER_CURSOR procedure associates the given name with the given cursor information. This name can then be used with the IDLgrWindow::SetCurrentCursor method.
SHMDEBUG	It can be difficult to know when a variable created with the SHMVAR function loses its reference to the underlying memory segment created by SHMMAP. The SHMDEBUG function is used to enable a debugging mode in which IDL prints an informational message including a traceback every time such a variable loses its reference to the underlying segment. SHMDEBUG returns the previous setting of the debugging state.
SHMMAP	The SHMMAP procedure maps anonymous shared memory, or local disk files, into the memory address space of the currently executing IDL process. Mapped memory segments are associated with an IDL array specified by the user as part of the call to SHMMAP. The type and dimensions of the specified array determine the length of the memory segment.

Table 1-13: New Routines in IDL 5.6 (Continued)

New Routine	Description
SHMUNMAP	The SHMUNMAP procedure is used to remove a memory segment previously created by SHMMAP from the system. If the segment has no variables currently accessing it (that is, if its reference count is zero) the segment is immediately removed from the system. If the segment has variables still referencing it, the unmapping is delayed until the last such variable drops its reference.
SHMVAR	The SHMVAR function creates an IDL array variable that uses the memory from a current mapped memory segment created by the SHMMAP procedure. Variables created by SHMVAR are used in much the same way as any other IDL variable, and provide the IDL user with the ability to alter the contents of anonymous shared memory or memory mapped files.
SKIP_LUN	The SKIP_LUN procedure reads data in an open file and moves the file pointer. It is useful in situations where it is necessary to skip over a known amount of data in a file without the requirement of having the data available in an IDL variable.
SWAP_ENDIAN_INPLACE	The SWAP_ENDIAN_INPLACE procedure reverses the byte ordering of arbitrary scalars, arrays or structures. It can make “big endian” number “little endian” and vice-versa. Note that the BYTEORDER procedure can be used to reverse the byte ordering of *scalars and arrays* (SWAP_ENDIAN_INPLACE also allows structures).

Table 1-13: New Routines in IDL 5.6 (Continued)

New Routine	Description
TRUNCATE_LUN	The TRUNCATE_LUN procedure truncates the contents of a file open for write access at the current position of the file pointer. After this operation, all data before the current file pointer remains intact, and all data following the file pointer are gone. The position of the current file pointer is not altered.
WIDGET_COMBOBOX	The WIDGET_COMBOBOX function creates "combobox" widgets, which are similar to "droplist" widgets. The main difference between the combobox widget and the droplist widget is that the text field of the combobox can be made editable, allowing the user to enter a value that is not on the list.
WIDGET_TAB	The WIDGET_TAB function is used to create a tab widget. Tab widgets present a display area on which different "pages" (base widgets and their children) can be displayed by selecting the appropriate tab.
WIDGET_TREE	The WIDGET_TREE function is used to create and populate a tree widget. The tree widget presents a hierarchical view that can be used to organize a wide variety of data structures and information.

Table 1-13: New Routines in IDL 5.6 (Continued)

IDL Routine Enhancements

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

ATAN

Keyword or item	Description
PHASE	If this keyword is set, and the argument is a complex number Z, then the complex phase angle is computed as ATAN(Imaginary(Z), Real_part(Z)). If this keyword is not set then the complex arctangent is computed as described above. If the argument is not complex, or if two arguments are present, then this keyword is ignored. Tip - Using the PHASE keyword is equivalent to computing ATAN(Imaginary(Z), Real_part(Z)), but uses less memory and is faster.

BESELI, BESELJ, BESELK, BESELY

Keyword or item	Description
DOUBLE	Set this keyword equal to one to return a double-precision result, or to zero to return a single-precision result. The computations will always be done using double precision. The default is to return a single-precision result if both inputs are single precision, and to return a double-precision result in all other cases.
ITER	Set this keyword equal to a named variable that will contain the number of iterations performed. If the routine converged, the stored value will be equal to the order *N*. If *X* or *N* are arrays, ITER will contain a scalar representing the maximum number of iterations. Note - If the routine did not converge for an element of *X*, the corresponding element of the *Result* array will be set to the IEEE floating-point value NaN, and ITER will contain the largest order that *would have converged* for that *X* value.

BETA

Keyword or item	Description
Complex input arguments	The BETA function now accepts complex arguments.

COMPILE_OPT

Keyword or item	Description
STRICTARRSUBS Option	Specifying STRICTARRSUBS will cause IDL to treat out-of-range array subscripts within the body of the routine containing the COMPILE_OPT statement as an error.

CURVEFIT

Keyword or item	Description
YERROR	Set this keyword to a named variable that will contain the standard error between YFIT and Y.

DIGITAL_FILTER

Keyword or item	Description
DOUBLE	Set this keyword to use double-precision for computations and to return a double-precision result. Set DOUBLE=0 to use single-precision for computations and to return a single-precision result. The default is /DOUBLE if the Flow input is double precision, otherwise the default is DOUBLE=0.

ERF

Keyword or item	Description
Complex input argument	The ERF function now accepts complex arguments.

ERFC

Keyword or item	Description
Complex Input argument	The ERFC function now accepts complex arguments.

ERFCX

Keyword or item	Description
Complex input argument	The ERFCX function now accepts complex arguments.

EXPINT

Keyword or item	Description
ITER	Set this keyword equal to a named variable that will contain the actual number of iterations performed.

FILE_DELETE

Keyword or item	Description
ALLOW_NON-EXISTENT	If set, FILE_DELETE will quietly ignore attempts to delete a non-existent file. Other errors will still be reported. The QUIET keyword can be used instead to suppress all errors.
VERBOSE	The VERBOSE keyword causes FILE_DELETE to issue an informative message for every file it deletes.

GAMMA

Keyword or item	Description
Complex input argument	The GAMMA function now accepts complex arguments.

GAUSSFIT

Keyword or item	Description
CHISQ	Set this keyword to a named variable that will contain the value of the chi-square goodness-of-fit.
SIGMA	Set this keyword to a named variable that will contain the 1-sigma error estimates of the returned parameters.
YERROR	Set this keyword to a named variable that will contain the standard error between YFIT and Y.

HELP

Keyword or item	Description
SHARED_MEMORY	Set this keyword to display information about all current shared memory and memory mapped file segments mapped into the current IDL process via the SHMMAP procedure.

HISTOGRAM

Keyword or item	Description
LOCATIONS	Set this keyword to a named variable in which to return the starting locations for each bin. The starting locations are given by MIN + *v**BINSIZE, with *v* = 0,1,...,NBINS-1. LOCATIONS has the same number of elements as the Result, and has the same type as the input Array.

IBETA

Keyword or item	Description
A	*A* may now be complex.

Keyword or item	Description
B	*B* may now be complex.
Z	*Z* may now be complex. If *Z* is not complex then the values must be in the range [0, 1].

IGAMMA

Keyword or item	Description
A	*A* may now be complex.
Z	*Z* may now be complex. If *Z* is not complex then the values must be greater than or equal to zero.

ISOCONTOUR

Keyword or item	Description
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_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 (with the corresponding label information returned via the OUT_LABEL_POLYS, OUT_LABEL_OFFSETS, and OUT_LABEL_STRINGS keywords). 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.

Keyword or item	Description
OUT_LABEL_OFFSETS	Set this keyword to a named variable that upon return will contain a vector of offsets (parameterized to the corresponding contour line) indicating the positions of the contour labels. Note - The C_LABEL_SHOW keyword should be specified if this keyword is used.
OUT_LABEL_POLYLINES	Set this keyword to a named variable that upon return will contain a vector of polyline indices, [P0, P1, …], that indicate which contour lines are labeled. Pi corresponds to the ith polyline specified via the Outconn argument. Note that if a given contour line has more than one label along its perimeter, then the corresponding polyline index may appear more than once in the LABEL_POLYS vector. Note - The C_LABEL_SHOW keyword should be specified if this keyword is used.
OUT_LABEL_STRINGS	Set this keyword to a named variable that upon return will contain a vector of strings, [str0, str1, …], that indicate the label strings. Note - The C_LABEL_SHOW keyword should be specified if this keyword is used.

KEYWORD_SET

Keyword or item	Description
Return value	The KEYWORD_SET function returns True (1) if: • *Expression* is a scalar or 1-element array with a non-zero value. • *Expression* is a structure. • *Expression* is an ASSOC file variable. KEYWORD_SET returns False (0) if *Expression* is undefined, or is a scalar or 1-element array with a zero value.

LNGAMMA

Keyword or item	Description
Complex input argument	The LNGAMMA function now accepts complex arguments.

MAKE_DLL

Keyword or item	Description
REUSE_EXISTING	If this keyword is set, and the sharable library file specified by *OutputFile* already exists, MAKE_DLL returns without building the sharable library again. Use this keyword in situations where you wish to ensure that a library exists, but only want to build it if it does not. Combining the REUSE_EXISTING and DLL_PATH keywords allows you to get a path to the library in a platform independent manner, building the library only if necessary.

MEDIAN

Keyword or item	Description
DIMENSION	Set this keyword to the dimension over which to find the median values of an array. If this keyword is not present or is zero, the median is found over the entire array and is returned as a scalar value. If this keyword is present and nonzero, the result is a “slice” of the input array that contains the median value elements, and the return value will be an array of one dimension less than the input array.

SVDFIT

Keyword or item	Description
SING_VALUES	Set this keyword to a named variable in which to return the singular values from the SVD. Singular values which have been removed will be set to zero.
STATUS	Set this keyword to a named variable that will contain the status of the computation. Possible values are: • STATUS = 0: The computation was successful. • STATUS > 0: Singular values were found and were removed. STATUS contains the number of singular values. Note - If STATUS is not specified, any error messages will be output to the screen.
TOL	Set this keyword to the tolerance used when removing singular values. The default is 10^{-5} for single precision, and $2x10^{-12}$ for double precision (these defaults are approximately 100 and 10000 times the machine precisions for single and double precision, respectively). Setting TOL to a larger value may remove coefficients that do not contribute to the solution, which may reduce the errors on the remaining coefficients.

SWAP_ENDIAN

Keyword or item	Description
SWAP_IF_BIG_ENDIAN	If this keyword is set, the swap request will only be performed if the platform running IDL uses “big endian” byte ordering. On little endian machines, the SWAP_ENDIAN request quietly returns without doing anything. Note that this keyword does not refer to the byte ordering of the input data, but to the computer hardware.

Keyword or item	Description
SWAP_IF_LITTLE_ ENDIAN	If this keyword is set, the swap request will only be performed if the platform running IDL uses “little endian” byte ordering. On big endian machines, the SWAP_ENDIAN request quietly returns without doing anything. Note that this keyword does not refer to the byte ordering of the input data, but to the computer hardware.

WIDGET_BASE

Keyword or item	Description
TLB_ICONIFY_EVENTS	Set this keyword when creating a top-level base to make that base return an event when the base is iconified or restored by the user.
TLB_MOVE_EVENTS	Set this keyword when creating a top-level base to make that base return an event when the base is moved on the screen by the user.
TOOLBAR	Set this keyword to indicate that the base is used to hold bitmap buttons that make up a toolbar. Note - Setting this keyword does not cause any changes in behavior; its only affect is to slightly alter the appearance of the bitmap buttons on the base for cosmetic reasons. Note - On Motif platforms, if bitmap buttons are on a toolbar base that is also EXCLUSIVE or NONEXCLUSIVE, they will not have a separate “toggle” indicator, they will be grouped closely together, and will have a two-pixel shadow border. Note - This keyword has no effect on Windows platforms.

Keyword or item	Description
Iconify Event Structure	Top-level widget bases return the following event structure when the base is iconified or restored and the base was created with the TLB_ICONIFY_EVENTS keyword set: `{ WIDGET_TLB_ICONIFY, ID:0L, TOP:0L, HANDLER:0L, ICONIFIED:0 }` ID is the widget ID of the base generating the event. TOP is the widget ID of the top level widget containing ID. HANDLER contains the widget ID of the widget associated with the handler routine. ICONIFIED is 1 (one) if the user iconified the base and 0 (zero) if the user restored the base.
Move Event Structure	Top-level widget bases return the following event structure when the base is moved and the base was created with the TLB_MOVE_EVENTS keyword set: `{ WIDGET_TLB_MOVE, ID:0L, TOP:0L, HANDLER:0L, X:0L, Y:0L }` ID is the widget ID of the base generating the event. TOP is the widget ID of the top level widget containing ID. HANDLER contains the widget ID of the widget associated with the handler routine. X and Y are the new location of the top left corner of the base. Note - On Windows, move events are generated while dragging. On UNIX, move events are generated only on the mouse-up. Note - If both TLB_SIZE_EVENTS and TLB_MOVE_EVENTS are enabled, a user resize operation that causes the top left corner of the base to move will generate both a move event and a resize event.

WIDGET_BUTTON

Keyword or item	Description
CHECKED_MENU	Set this keyword on a menu entry button to enable the ability to place a check or selection box next to the menu entry. The parent widget of the button must be either a button widget created with the MENU keyword or a base widget created with the CONTEXT_MENU keyword.
TOOLTIP	Set this keyword to a string that will be displayed when the cursor hovers over the widget. For UNIX platforms, this string must be non-zero in length.

WIDGET_CONTROL

Keyword or item	Description
COMBOBOX_ ADDITEM	This keyword applies to widgets created with the WIDGET_COMBOBOX function. Set this keyword to a string that specifies a new item to add to the list of the combobox. By default, the item will be added to the end of the list. The item can be added to a specified position in the list by setting the COMBOBOX_INDEX keyword in the same call to WIDGET_CONTROL.
COMBOBOX_ DELETEITEM	This keyword applies to widgets created with the WIDGET_COMBOBOX function. Set this keyword to an integer that specifies the zero-based index of the combobox element to be deleted from the list. If the specified element is outside the range of existing elements, no element is deleted.

Keyword or item	Description
COMBOBOX_ INDEX	This keyword applies to widgets created with the WIDGET_COMBOBOX function. Set this keyword to an integer that specifies the zero-based index at which a new item will be added to the list when using the COMBOBOX_ADDITEM keyword. If the supplied index is outside the range of zero to the length of the existing list, the item is not added to the list. Note - You can retrieve the length of the existing list using the COMBOBOX_NUMBER keyword to WIDGET_INFO.
DRAW_KEY-BOARD_EVENTS	This keyword applies to widgets created with the WIDGET_DRAW function. Set this keyword equal to 1 (one) or 2 to make the draw widget generate an event when it has the keyboard focus and a key is pressed or released. (The method by which a widget receives the keyboard focus is dependent on the window manager in use.) The value of the key pressed is reported in either the `CH` or the `KEY` field of the event structure, depending on the type of key pressed. • If this keyword is set equal to 1, the draw widget will generate an event when a “normal” key is pressed. “Normal” keys include all keys except function keys and the modifier keys: SHIFT, CONTROL, CAPS LOCK, and ALT. If a modifier key is pressed at the same time as a normal key, the value of the modifier key is reported in the `MODIFIERS` field of the event structure. • If this keyword is set equal to 2, the draw widget will generate an event when *either* a normal key or a modifier key is pressed. Values for modifier keys are reported in the `KEY` field of the event structure, and the `MODIFIERS` field contains zero. Note - Keyboard events are never generated for function keys.

Keyword or item	Description
SET_BUTTON	This keyword applies to widgets created with the WIDGET_BUTTON function. This keyword changes the current state of toggle buttons. If set equal to zero, every toggle button in the hierarchy specified by *Widget_ID* is set to the unselected state. If set to a nonzero value, the action depends on the type of base holding the buttons.
SET_COMBOBOX_ SELECT	This keyword applies to widgets created with the WIDGET_COMBOBOX function. Set this keyword to an integer that specifies the zero-based index of the combobox list element to be displayed. If the specified element is outside the range of existing elements, the selection remains unchanged.
SET_TAB_ CURRENT	This keyword applies to widgets created with the WIDGET_TAB function. Set this keyword equal to the zero-based index of the tab to be set as the current (visible) tab. If the index value is invalid, the value is quietly ignored.

Keyword or item	Description
SET_TAB_ MULTILINE	This keyword applies to widgets created with the WIDGET_TAB function. This keyword controls how tabs appear on the tab widget when all of the tabs do not fit on the widget in a single row. This keyword behaves differently on Windows and Motif systems. **Windows** Set this keyword to cause tabs to be organized in a multi-line display when the width of the tabs exceeds the width of the largest child base widget. If possible, IDL will create tabs that display the full tab text. If MULTILINE=0 and LOCATION=0 or 1, tabs that exceed the width of the largest child base widget are shown with scroll buttons, allowing the user to scroll through the tabs while the base widget stays immobile. If LOCATION=1 or 2, a multiline display is always used if the tabs exceed the height of the largest child base widget. **UNIX** Set this keyword equal to an integer that specifies the maximum number of tabs to display per row in the tab widget. If this keyword is not specified (or is explicitly set equal to zero) all tabs are placed in a single row.
SET_TREE_ BITMAP	This keyword applies to widgets created with the WIDGET_TREE function. Set this keyword equal to a 16x16x3 array representing an RGB image that will be displayed next to the node in the tree widget. Set this keyword equal to zero to revert to the appropriate default system bitmap.
SET_TREE_ EXPANDED	This keyword applies to widgets created with the WIDGET_TREE function. Set this keyword equal to a nonzero value to expand the specified tree widget folder. Set this keyword equal to zero to collapse the specified tree widget folder.

Keyword or item	Description
SET_TREE_SELECT	This keyword applies to widgets created with the WIDGET_TREE function. This keyword has two modes of operation, depending on the widget ID passed to WIDGET_CONTROL. If the specified widget ID is for the root node of the tree widget (the tree widget whose *Parent* is a base widget): • If the tree widget is in multiple-selection mode and SET_TREE_SELECT is set to an array of widget IDs corresponding to tree widgets that are nodes in the tree, those nodes are selected. • If the tree widget is *not* in multiple-selection mode and SET_TREE_SELECT is set to a single widget ID corresponding to a tree widget that is a node in the tree, that node is selected. • If the keyword is set to zero, all selections in the tree widget are cleared. If the specified widget ID is a tree widget that is a node in a tree: • If the keyword is set to a nonzero value, the specified node is selected. • If the keyword is set to zero, the specified node is deselected. Note - If the tree widget is in multiple-selection mode, the selection changes made to the tree widget via this keyword are additive — that is, the current selections are retained and any additional nodes specified by SET_TREE_SELECT are also selected.

Keyword or item	Description
SET_TREE_ VISIBLE	This keyword applies to widgets created with the WIDGET_TREE function and whose parent widget was also created using the WIDGET_TREE function (that is, tree widgets that are nodes of another tree). Set this keyword to make the specified tree node *visible* to the user. Setting this keyword has two possible effects: 1. If the specified node is inside a collapsed folder, the folder and all folders above it are expanded to reveal the node. 2. If the specified node is in a portion of the tree that is not currently visible because the tree has scrolled within the parent base widget, the tree view scrolls so that the selected node is at the top of the base widget. Use of this keyword does not affect the tree widget selection state.
TABLE_BLANK	This keyword applies to widgets created with the WIDGET_TABLE function. Set this keyword equal to a nonzero value to cause the specified cells to be blank. Set this keyword equal to zero to cause the specified cells to display values as usual. If the USE_TABLE_SELECT keyword is set equal to one, the currently selected cells are blanked or restored. If USE_TABLE_SELECT is set equal to an array, the specified cells are blanked or restored. If USE_TABLE_SELECT is not set, the entire table is blanked or restored.
TABLE_DISJOINT_ SELECTION	This keyword applies to widgets created with the WIDGET_TABLE function. Set this keyword to enable the ability to select multiple rectangular regions of cells.
TLB_ICONIFY_ EVENTS	This keyword applies to widgets created with the WIDGET_BASE function. Set this keyword to make the top-level base return an event when the base is iconified or restored by the user.

Keyword or item	Description
TLB_MOVE_ EVENTS	This keyword applies to widgets created with the WIDGET_BASE function. Set this keyword to make the top-level base return an event when the base is moved by the user. Note that if TLB_SIZE_EVENTS are also enabled, a user resize operation that causes the top left corner of the base widget to move will generate both a move event and a resize event.
TLB_SIZE_ EVENTS	This keyword applies to widgets created with the WIDGET_BASE function. Set this keyword to make the top-level base return an event when the base is resized by the user. Note that if TLB_MOVE_EVENTS are also enabled, a user resize operation that causes the top left corner of the base widget to move will generate both a move event and a resize event.
TOOLTIP	This keyword applies to widgets created with the WIDGET_BUTTON and WIDGET_DRAW functions. Set this keyword to a string that will be displayed when the cursor hovers over the specified widget. For UNIX platforms, this string must be non-zero in length, which means that a tooltip can be modified but not be removed on UNIX versions of IDL.

WIDGET_DRAW

Keyword or item	Description
KEYBOARD_EVENTS	Set this keyword equal to 1 (one) or 2 to make the draw widget generate an event when it has the keyboard focus and a key is pressed or released. (The method by which a widget receives the keyboard focus is dependent on the window manager in use.) The value of the key pressed is reported in either the `CH` or the `KEY` field of the event structure, depending on the type of key pressed. • If this keyword is set equal to 1, the draw widget will generate an event when a "normal" key is pressed. "Normal" keys include all keys except function keys and the modifier keys: SHIFT, CONTROL, CAPS LOCK, and ALT. If a modifier key is pressed at the same time as a normal key, the value of the modifier key is reported in the `MODIFIERS` field of the event structure. • If this keyword is set equal to 2, the draw widget will generate an event when *either* a normal key or a modifier key is pressed. Values for modifier keys are reported in the `KEY` field of the event structure, and the `MODIFIERS` field contains zero. Note - Keyboard events are never generated for function keys.
TOOLTIP	Set this keyword to a string that will be displayed when the cursor hovers over the widget. For UNIX platforms, this string must be non-zero in length.

WIDGET_INFO

Keyword or item	Description
BUTTON_SET	This keyword applies to widgets created with the WIDGET_BUTTON function. Set this keyword to return the “set” state of a widget button. If the button is currently set, 1 (one) is returned. If the button is currently not set, 0 (zero) is returned. This keyword is intended for use with exclusive, non-exclusive and checked menu buttons.
COMBOBOX_GETTEXT	This keyword applies to widgets created with the WIDGET_COMBOBOX function. Set this keyword to return the current text from the text box of the specified combobox widget. Note that when using an editable combobox, the text displayed in the text box may not be an item from the list of values in the combobox list. To obtain the index of the selected item, inspect the INDEX field of the event structure returned by the combobox widget.
COMBOBOX_NUMBER	This keyword applies to widgets created with the WIDGET_COMBOBOX function. Set this keyword to return the number of elements currently contained in the list of the specified combobox widget.
FONTNAME	This keyword applies to all widgets. Set this keyword to return a string containing the name of the font being used by the specified widget. The returned name can then be used when creating other widgets or with the SET_FONT keyword to the DEVICE procedure.

Keyword or item	Description
MAP	This keyword applies to all widgets. Set this keyword to return True (1) if the widget specified by *Widget_ID* is mapped (visible), or False (0) otherwise. Note that when a base widget is unmapped, all of its children are unmapped. If WIDGET_INFO reports that a particular widget is unmapped, it may be because a parent in the widget hierarchy has been unmapped.
SENSITIVE	This keyword applies to all widgets. Set this keyword to return True (1) if the widget specified by *Widget_ID* is sensitive (enabled), or False (0) otherwise. Note that when a base is made insensitive, all its children are made insensitive. If WIDGET_INFO reports that a particular widget is insensitive, it may be because a parent in the widget hierarchy has been made insensitive.
TAB_CURRENT	This keyword applies to widgets created with the WIDGET_TAB function. Set this keyword to return the zero-based index of the current tab in the tab widget.
TAB_MULTILINE	This keyword applies to widgets created with the WIDGET_TAB function. Set this keyword to return the current setting of the multi-line mode for the tab widget.
TAB_NUMBER	This keyword applies to widgets created with the WIDGET_TAB function. Set this keyword to return the number of tabs contained in the tab widget.
TABLE_DISJOINT_ SELECTION	This keyword applies to widgets created with the WIDGET_TABLE function. Set this keyword to return 1 (one) if the widget specified by *Widget_ID* has disjoint selection enabled. Otherwise, 0 (zero) is returned.

Keyword or item	Description
TLB_ICONIFY_EVENTS	This keyword applies to widgets created with the WIDGET_BASE function. Set this keyword to return 1 if the top-level base widget specified by *Widget_ID* is set to return iconify events. Otherwise, 0 is returned.
TLB_MOVE_EVENTS	This keyword applies to widgets created with the WIDGET_BASE function. Set this keyword to return 1 if the top-level base widget specified by *Widget_ID* is set to return move events. Otherwise, 0 is returned.
TLB_SIZE_EVENTS	This keyword applies to widgets created with the WIDGET_BASE function. Set this keyword to return 1 if the top-level base widget specified by *Widget_ID* is set to return resize events. Otherwise, 0 is returned.
TOOLTIP	This keyword applies to widgets created with the WIDGET_BUTTON and WIDGET_DRAW functions. Set this keyword to have the WIDGET_INFO function return the text of the tooltip of the widget. If the widget does not have a tooltip, a null string will be returned.
TREE_EXPANDED	This keyword applies to widgets created with the WIDGET_TREE function. Set this keyword to return 1 (one) if the specified tree widget node is a folder that is expanded, or 0 (zero) if the specified node is a folder that is collapsed. Note - Only tree widget nodes created with the FOLDER keyword can be expanded or collapsed. This keyword will always return 0 (zero) if the specified tree widget node is not a folder.

Keyword or item	Description
TREE_ROOT	This keyword applies to widgets created with the WIDGET_TREE function. Set this keyword to return the widget ID of the *root node* of the tree widget hierarchy of which *Widget ID* is a part. The root node is the tree widget whose parent is a base widget.
TREE_SELECT	This keyword applies to widgets created with the WIDGET_TREE function. Set this keyword to return information about the nodes selected in the specified tree widget. This keyword has two modes of operation, depending on the widget ID passed to WIDGET_INFO: • If the specified widget ID is for the root node of the tree widget (the tree widget whose *Parent* is a base widget), this keyword returns either the widget ID of the selected node or (if multiple nodes are selected) an array of widget IDs of the selected nodes. If no nodes are selected, -1 is returned. • If the specified widget ID is a tree widget that is a node in a tree, this keyword returns 1 (one) if the node is selected or 0 (zero) if it is not selected.
VISIBLE	This keyword applies to all widgets. Set this keyword to return True (1) if the widget specified by *Widget_ID* is visible, or False (0) otherwise. A widget is visible if: • it has been realized, • it and all of its ancestors are mapped.

WIDGET_LABEL

Keyword or item	Description
SUNKEN_FRAME	Set this keyword to create a three dimensional, bevelled border around the label widget. The resulting frame gives the label a “sunken” appearance, similar to what is often seen in application status bars.

WIDGET_TABLE

Keyword or item	Description
DISJOINT_SELECTION	Set this keyword to enable the ability to select multiple rectangular regions of cells. The regions can be overlapping, touching, or entirely distinct. Setting this keyword changes the data structures returned by the TABLE_SELECT keyword to WIDGET_INFO and the GET_VALUE keyword to WIDGET_CONTROL. Similarly, the data structures you supply via the SET_TABLE_SELECT and SET_VALUE keywords to WIDGET_CONTROL are different in disjoint mode.

WRITE_TIFF

Keyword or item	Description
COMPRESSION	Set this keyword to select the type of compression to be used: • 0 = none (default) • 2 = PackBits • 3 = JPEG (ITIFF files)

XROI

Keyword or item	Description
X_SCROLL_SIZE	Set this keyword to the width of the scroll window. If this keyword is larger than the image width then it will be set to the image width. The default is to use the image width or the screen width, whichever is smaller.
Y_SCROLL_SIZE	Set this keyword to the height of the scroll window. If this keyword is larger than the image height then it will be set to the image height. The default is to use the image height or the screen height, whichever is smaller.

ION 1.6 Enhancements

ION (*IDL On the Net*) is now released with IDL. ION Script and ION Java are packages for publishing IDL-driven applications on the Web. They are now included on the IDL CD as an optional feature. An extra-cost ION license is required to use ION Script and ION Java. For more information on ION, see "Introduction to ION" in the ION manual.

ION Script Enhancements

This section discusses the following new features and enhancements in ION Script 1.6:

- New ION_OBJECT Tag
- New FORMAT Attribute For ION Script Variables
- ION_EVALUATE and ION_VARIABLE Can Now Be Used Inside <IDL> Blocks
- New Support For MULTIPLE Attribute In HTML SELECT Tag
- New Example For Passing Data From IDL to ION Script

New ION_OBJECT Tag

In ION Script 1.4, it was only possible to embed IDL-generated text (via ION_DATA_OUT) and 2-D images (via ION_IMAGE) in your Web application. IDL is capable of generating (and Web servers are capable of passing) numerous types of data, such as VRML, MPEG, and WAV. With the addition of the ION_OBJECT tag, ION_PARAM tag, and $ION.IDLURL system variable in ION 1.6, it is now possible to embed in your Web application any type of data that IDL can generate or locate.

For more information and examples, see "ION_OBJECT" in Chapter 5 of the *ION Script User's Guide* manual.

New FORMAT Attribute For ION Script Variables

The ION_EVALUATE and ION_VARIABLE tags now support a FORMAT attribute, which allows you to specify the format of your variable using a C-style printf() format specifier.

For more information, see the description of the FORMAT attribute for "ION_EVALUATE" or "ION_VARIABLE" in Chapter 5 of the *ION Script User's Guide* manual.

ION_EVALUATE and ION_VARIABLE Can Now Be Used Inside <IDL> Blocks

In ION 1.4, the only part of an <IDL> block that was evaluated by the ION Script parser before sending the data to IDL was ION Script variables. In ION 1.6, you can now include ION_EVALUTE and ION_VARIABLE tags inside an <IDL> block. These tags are first evaluated by the parser, then ION Script variables are evaluated. This allows you to format ION Script variables before sending them to IDL.

See “Using ION_EVALUTE and ION_VARIABLE Tags in an IDL Block” in Chapter 5 of the *ION Script User’s Guide* manual for an example.

New Support For MULTIPLE Attribute In HTML SELECT Tag

When the MULTIPLE attribute is specified for the HTML <SELECT> tag, the user is allowed to select multiple options. Suppose a user submits a form after selecting the following options in a SELECT element named Region:

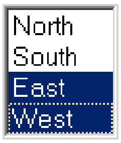


Figure 1-9: Example of Using the MULTIPLE Attribute

When the user submits the ION_FORM, the URL sent to the server takes the following form:

```
http://host/cgi-bin/ion-p.exe?Region=East&Region=West
```

In ION 1.4, the $Form variable created on the page that is loaded when this form is submitted would contain the value “West” (the last value for Region specified in the query string). In ION 1.6, the $Form variable created for a SELECT element contains the value of all selected options, separated by the “|” character. Therefore, in the above example, the value of the $Form variable $Form.Region would be “East|West” if the user selected the “East” and “West” options and submitted the form.

For more information and examples, see “Handling Multiple Selections in a SELECT Element” in Chapter 4 of the *ION Script User’s Guide* manual.

New Example For Passing Data From IDL to ION Script

A new example illustrating how to pass data from IDL to ION Script has been added to the Advanced examples page. Load the page index_examples.ion, and click the "Passing IDL Variables to ION Script Example" link.

ION Java Enhancements

This section discusses the following new features and enhancements in ION Java 1.6:

- IONGr2Canvas Class Now Obsolete
- IDL Command Execution Status Now Properly Reported
- New IONVariable Methods Return Dimensioned Results
- New Supported Keywords for Contours, Maps, Plots, and Surfaces

IONGr2Canvas Class Now Obsolete

Because there is a more efficient method for accessing IDL Object Graphics in ION Java, the IONGr2Canvas class has been obsoleted. See "Object Graphics in ION" in Chapter 5 of the *ION Java User's Guide* manual for details on how to use Object Graphics in ION Java.

You can also run the Object Graphics example on the `advanced.html` page in the `idl56\products\ion16\ion_java\examples` directory (Windows) or `idl_5.6/products/ion_1.6/ion_java/examples` directory (UNIX) of your IDL installation.

IDL Command Execution Status Now Properly Reported

The executeIDLCommand() method of the IONCallableClient, IONGrConnection/IONJGrConnection, and IONGrDrawable/IONJGrDrawable classes have been fixed so that they properly return the execution status of IDL commands. If the IDL command executes successfully, the executeIDLCommand() method returns 0. If the IDL command does not execute successfully, the executeIDLCommand() method returns the value of the !ERROR IDL system variable.

New IONVariable Methods Return Dimensioned Results

In prior versions of ION Java, the getByteArray(), getComplexArray(), getDoubleArray(), getFloatArray(), getIntArray(), getShortArray(), and getStringArray() methods of the IONVariable class were used to get arrays for the specified variable. The result of these methods is a one-dimensional array, even if the variable contains two or more dimensions. Using these methods, the ION Java programmer must reformat the array into the proper number of dimensions.

The following new methods of IONVariable are provided to eliminate the need to manually reformat the array. These methods return an array with the same number of dimensions as the variable:

- getDimensionedByteArray()
- getDimensionedDoubleArray()
- getDimensionedFloatArray()
- getDimensionedIntArray()
- getDimensionedShortArray()

New Supported Keywords for Contours, Maps, Plots, and Surfaces

The IONGrContour, IONGrMap, IONGrMapContinents, IONGrMapGrid, IONGrMapImage, IONGrPlot, and IONGrSurface classes now support additional keywords of the IDL CONTOUR, MAP_SET, MAP_CONTINENTS, MAP_GRID, MAP_IMAGE, PLOT, and SURFACE procedures, respectively.

See the following sections in Chapter 6, “ION Java Class and Method Reference” in the *ION Java User’s Guide* manual for a list of the keywords supported by each class:

- IONGrContour—Properties Supported
- IONGrMap—Properties Supported
- IONGrMapContinents—Properties Supported
- IONGrMapGrid—Properties Supported
- IONGrMapImage—Properties Supported
- IONGrPlot—Properties Supported
- IONGrSurface—Properties Supported

Routines Obsoleted in IDL 5.6

The following routines were present in IDL Version 5.5 but became obsolete in Version 5.6. 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
VAX_FLOAT	VAX_FLOAT keyword to OPEN
HDF_VD_GETNEXT	HDF_VG_GETNEXT

Requirements for this Release

IDL 5.6 Requirements

Hardware Requirements for IDL 5.6

The following table describes the supported platforms and operating systems for IDL 5.6:

Platform	Vendor	Hardware	Operating System	Supported Versions
Windows	Microsoft	Intel x86	Windows	98
		Intel x86	Windows NT	4.0, 2000, XP
Macintosh	Apple	PowerMac G4	Mac OS X	10.1, 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	Linux	Red Hat 7.1††
	SGI	Mips 32-bit	IRIX	6.5.1
	SGI	Mips 64-bit	IRIX	6.5.1
	SUN	SPARC 32-bit	Solaris	8
	SUN	SPARC 64-bit	Solaris	8

Table 1-14: Hardware Requirements for IDL 5.6.

On platforms that provide 64-bit support, IDL can be run as either a 32-bit or a 64-bit application. 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
```

† 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 5.6 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 IDL on your version.

Software Requirements for IDL 5.6

The following table describes the software requirements for IDL 5.6:

Platform	Software Requirements
Windows	Internet Explorer 5.0 or higher.
Macintosh	XFree86 version 4.2 (XDarwin 1.0.6.) which is included on the IDL 5.6 product CD.

Table 1-15: Software Requirements for IDL 5.6

ION 1.6 Requirements

Hardware Requirements for ION 1.6

The following table describes the supported platforms and operating systems for ION 1.6:

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

Table 1-16: Hardware Requirements for IDL 5.6.

† 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 1.6 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 1.6 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 4 for Windows NT 4.0 Server, 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 5.6 CD-ROM contains the following Apache Web Server software:

- Windows — Version 2.0.40
- Linux — Version 2.0.40
- Solaris — Version 2.0.39
- IRIX — Version 1.3.14

Note

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

Web Browsers

ION 1.6 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 1.6 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:

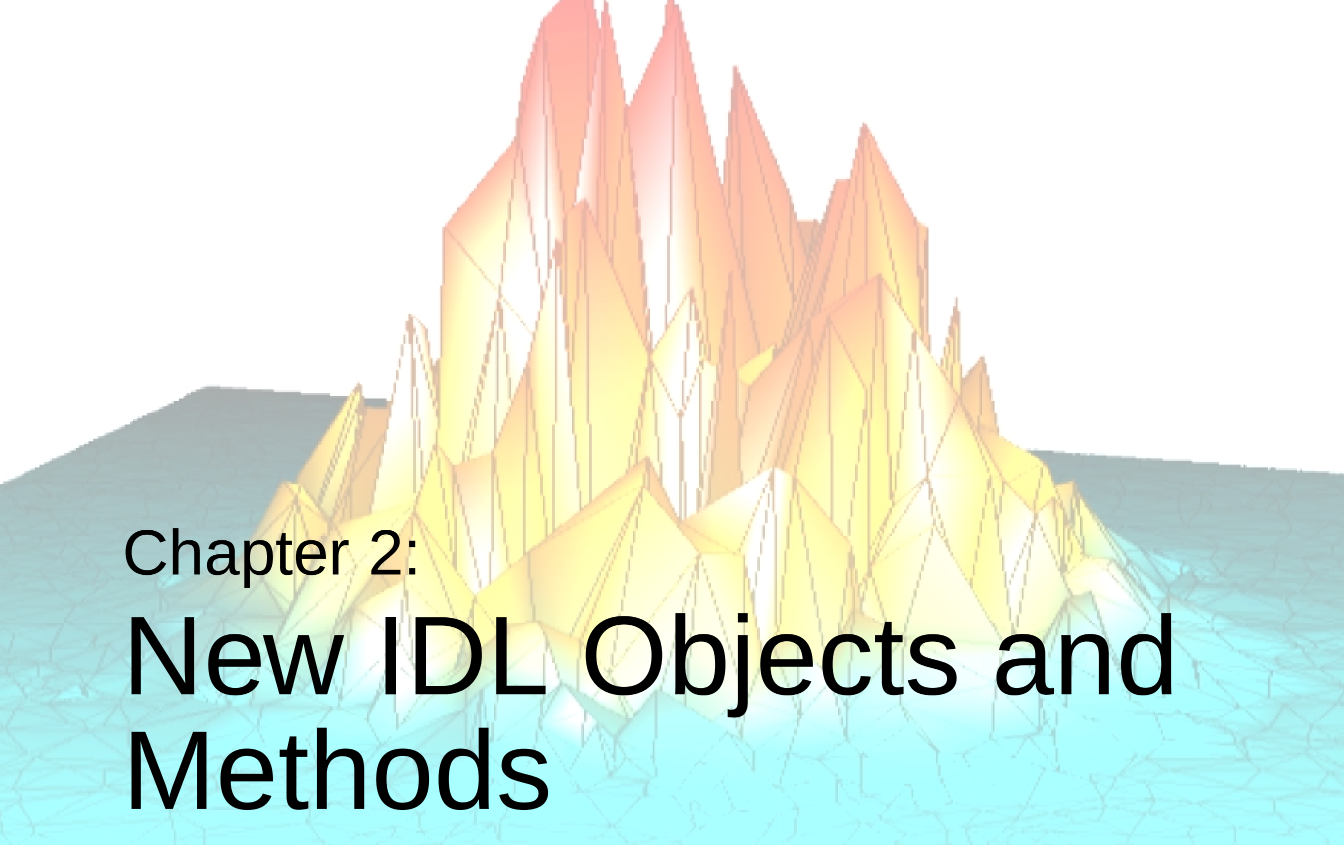
- Netscape Navigator versions 4.7 and 6.0.
- Microsoft Internet Explorer versions 5.5 and 6.0.

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.

Windows 98 Platform Support Ending

IDL 5.6 will be the last release to support the Windows 98 platform. We recommend that you consider upgrading to a later release of Microsoft Windows to be able to run future versions of IDL.

RSI is committed to supporting our customers with their varied platform requirements while maintaining financially sound business practices. Our goal is to communicate platform support plans in a timely fashion in order to allow you ample time to make well informed platform decisions.



Chapter 2:

New IDL Objects and Methods

This chapter describes new objects and new methods to existing objects introduced in IDL 5.6

IDLffXMLSAX object

An IDLffXMLSAX object uses an XML SAX level 2 parser. The XML parser allows you to read an XML file and store arbitrary data from the file in IDL variables. The parser object's methods are *callbacks*. These methods are called automatically when the parser encounters different types of XML elements or attributes.

Note

To use the XML parser, you *must* write a subclass of this object class, overriding the object methods as necessary to process the data in a specific XML file or files. See Chapter 22, "Using the XML Parser Object Class" in the *Building IDL Applications* manual for further information and examples.

The IDLffXMLSAX object encapsulates the Xerces validating XML parser; see `http://xml.apache.org` for details.

Superclasses

This class has no superclass.

Subclasses

You *must* write a subclass of this object, overriding object methods as necessary to retrieve information from the XML file.

Creation

See "IDLffXMLSAX::Init" on page 167

Methods

Intrinsic Methods

This class has the following methods:

- IDLffXMLSAX::AttributeDecl
- IDLffXMLSAX::Characters
- IDLffXMLSAX::Cleanup
- IDLffXMLSAX::Comment

- IDLffXMLSAX::ElementDecl
- IDLffXMLSAX::EndCDATA
- IDLffXMLSAX::EndDocument
- IDLffXMLSAX::EndDTD
- IDLffXMLSAX::EndElement
- IDLffXMLSAX::EndEntity
- IDLffXMLSAX::EndPrefixMapping
- IDLffXMLSAX::Error
- IDLffXMLSAX::ExternalEntityDecl
- IDLffXMLSAX::FatalError
- IDLffXMLSAX::GetProperty
- IDLffXMLSAX::IgnorableWhitespace
- IDLffXMLSAX::Init
- IDLffXMLSAX::InternalEntityDecl
- IDLffXMLSAX::NotationDecl
- IDLffXMLSAX::ParseFile
- IDLffXMLSAX::ProcessingInstruction
- IDLffXMLSAX::SetProperty
- IDLffXMLSAX::SkippedEntity
- IDLffXMLSAX::StartCDATA
- IDLffXMLSAX::StartDocument
- IDLffXMLSAX::StartDTD
- IDLffXMLSAX::StartElement
- IDLffXMLSAX::StartEntity
- IDLffXMLSAX::StartPrefixMapping
- IDLffXMLSAX::StopParsing
- IDLffXMLSAX::UnparsedEntityDecl
- IDLffXMLSAX::Warning

Version History

Introduced: 5.6

IDLffXMLSAX::AttributeDecl

The IDLffXMLSAX::AttributeDecl procedure method is called when the parser detects an `<!ATTLIST ...>` declaration in a DTD. This method is called once for each attribute declared by the tag.

Syntax

Obj `->` [IDLffXMLSAX::]AttributeDecl, *eName*, *aName*, *Type*, *Mode*, *Value*

Arguments

eName

A named variable that will contain the name of the element for which the attribute is being declared.

aName

A named variable that will contain the name of the attribute being declared.

Type

A named variable that will contain a string that specifies the type of attribute being defined. Possible values are:

- 'CDATA'
- 'ID'
- 'IDREF'
- 'IDREFS'
- 'NMTOKEN'
- 'NMTOKENS'
- 'ENTITY'
- 'ENTITIES'

or two types of enumerated values. Enumerated values are encoded with parenthesized strings such as `(a|b|c)` to indicate that strings `a`, `b`, or `c` are permissible. If the string is an enumeration of notation names, the string `"NOTATION "` (note the space after the second “N”) precedes the parenthesized string.

Mode

A named variable that will contain a string that specifies restrictions on the value of the attribute. Possible values are:

- '#IMPLIED' - the application determines the value
- '#REQUIRED' - the value must be given; defaulting is not permitted
- '#FIXED' - only one value is permitted
- '' - a null string (the value specified by the *Value* argument is used as the default)

Value

A named variable that will contain the default value for the attribute. If *Value* contains a null string, no default value was specified.

Keywords

None.

IDLffXMLSAX::Characters

The IDLffXMLSAX::Characters procedure method is called when the parser detects text in the parsed document.

Syntax

Obj -> [IDLffXMLSAX::]Characters, *Chars*

Arguments

Chars

A named variable that will contain the text detected by the parser.

Keywords

None.

IDLffXMLSAX::Cleanup

The IDLffXMLSAX::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object destruction. In most cases, you cannot call the Cleanup method directly. However, one exception to this rule does exist. If you write your own subclass of this class, you can call the Cleanup method from within the Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj `->` [IDLffXMLSAX::]Cleanup (*Only in subclass' Cleanup method*.)

Arguments

None.

Keywords

None.

IDLffXMLSAX::Comment

The IDLffXMLSAX::Comment procedure method is called when the parser detects a comment section of the form `<!-- ... -->`.

Syntax

Obj `->` [IDLffXMLSAX::]Comment, *Comment*

Arguments

Comment

A named variable that will contain the text within the detected comment section, without the delimiting characters (“`<!--`” and “`-->`”).

Keywords

None.

IDLffXMLSAX::ElementDecl

The IDLffXMLSAX::ElementDecl procedure method is called when the parser detects an `<!ELEMENT ...>` declaration in the DTD.

Syntax

Obj `->` [IDLffXMLSAX::]ElementDecl, *Name*, *Model*

Arguments

Name

A named variable that will contain the name of the element.

Model

A named variable that will contain the *content model* (sometimes called the *content specification*) for the element, with all whitespace removed.

Keywords

None.

IDLffXMLSAX::EndCDATA

The IDLffXMLSAX::EndCDATA procedure method is called when the parser detects the end of a `<[CDATA[...]]>` text section.

Syntax

Obj `->` [IDLffXMLSAX::]EndCDATA

Arguments

None.

Keywords

None.

IDLffXMLSAX::EndDocument

The IDLffXMLSAX::EndDocument procedure method is called when the parser detects the end of the XML document.

Syntax

Obj `->` [IDLffXMLSAX::]EndDocument

Arguments

None.

Keywords

None.

IDLffXMLSAX::EndDTD

The IDLffXMLSAX::EndDTD procedure method is called when the parser detects the end of a Document Type Definition (DTD).

Syntax

Obj `->` [IDLffXMLSAX::]EndDTD

Arguments

None.

Keywords

None.

IDLffXMLSAX::EndElement

The IDLffXMLSAX::EndElement procedure method is called when the parser detects the end of an element.

Syntax

Obj `->` [IDLffXMLSAX::]EndElement, *URI*, *Local*, *qName*

Arguments

URI

A named variable that will contain the namespace URI with which the element is associated, if any.

Note

A URI (or Uniform Resource Identifier) refers to the generic set of all names and addresses which are short strings which refer to objects.

Local

A named variable that will contain the element name with any prefix removed, if the element is associated with a namespace URI. If the element is not associated with a namespace URI, this variable will contain an empty string.

qName

A named variable that will contain the element name found in the XML file.

Note

If the element is associated with a namespace URI, this variable may contain an empty string.

Keywords

None.

IDLffXMLSAX::EndEntity

The IDLffXMLSAX::EndEntity procedure method is called when the parser detects the end of an internal or external entity expansion.

Syntax

Obj `->` [IDLffXMLSAX::]EndEntity, *Name*

Arguments

Name

A named variable that will contain the name of the entity.

Keywords

None.

IDLffXMLSAX::EndPrefixMapping

The IDLffXMLSAX::EndPrefixMapping procedure method is called when a previously declared prefix mapping goes out of scope.

Syntax

Obj `->` [IDLffXMLSAX::]EndPrefixMapping, *Prefix*

Arguments

Prefix

A named variable that will contain the namespace prefix that is going out of scope.

Keywords

None.

IDLffXMLSAX::Error

The IDLffXMLSAX::Error procedure method is called when the parser detects an error that is not expected to be fatal. This method prints an IDL error string to the IDL output log and allows the parser to continue processing.

For example, a violation of XML validity constraints is generally a non-fatal error.

Note

This method will cause error messages to be printed to the IDL output log. If you would like your application to hide error messages from the user (or display them in some other fashion), override this method in your subclass of the IDLffXMLSAX object class. If you do override this method, the error message will not be printed to the output log unless you explicitly call the superclass method.

Syntax

Obj -> [IDLffXMLSAX::]Error, *SystemID*, *LineNumber*, *ColumnNumber*, *Message*

Arguments

SystemID

A named variable that will contain the URI of the associated text.

LineNumber

A named variable that will contain the line number that contains the error.

ColumnNumber

A named variable that will contain the column number that contains the error.

Message

A named variable that will contain the error message sent to the IDL output log.

Keywords

None.

IDLffXMLSAX::ExternalEntityDecl

The IDLffXMLSAX::ExternalEntityDecl procedure method is called when the parser detects an `<!ENTITY ...>` declarations in the DTD for a parsed external entity.

Syntax

Obj `->` [IDLffXMLSAX::]ExternalEntityDecl, *Name*, *PublicID*, *SystemID*

Arguments

Name

A named variable that will contain the entity name.

PublicID

A named variable that will contain the Public ID for the entity.

Note

If this value is not specified in the entity declaration, this variable will contain an empty string.

SystemID

A named variable that will contain the System ID for the entity, provided as an absolute URI.

Keywords

None.

IDLffXMLSAX::FatalError

The IDLffXMLSAX::FatalError procedure method is called when the parser detects a fatal error. When called, parsing will normally stop, but may sometimes continue long enough to report further errors. This method prints an IDL error string to the IDL output log.

Syntax

Obj `->` [IDLffXMLSAX::]FatalError, *SystemID*, *LineNumber*, *ColumnNumber*, *Message*

Arguments

SystemID

A named variable that will contain the URI of the associated text.

LineNumber

A named variable that will contain the line number that contains the error.

ColumnNumber

A named variable that will contain the column number that contains the error.

Message

A named variable that will contain the error message sent to the IDL output log.

Keywords

None.

IDLffXMLSAX::GetProperty

The IDLffXMLSAX::GetProperty procedure method is used to get the values of various properties of the parser.

Syntax

Obj -> [IDLffXMLSAX::]GetProperty [, FILENAME=*variable*]
[, PARSER_LOCATION=*variable*] [, PARSER_PUBLICID=*variable*]
[, PARSER_URI=*variable*]

Arguments

None.

Keywords

Any keyword to the IDLffXMLSAX::Init followed by Get can be retrieved using IDLffXMLSAX::GetProperty. To retrieve a property, set the associated keyword equal to a named variable that will contain the property value.

In addition, the following keywords are available:

Note
These properties are only available during a parse operation.

FILENAME

Set this keyword equal to a named variable that will contain the filename of the XML file being parsed.

PARSER_LOCATION

Set this keyword equal to a named variable that will contain the approximate location of the parser within the entity being parsed. The value is returned as a two-element array, with the first element set to the line number and the second element set the column number.

PARSER_PUBLICID

Set this keyword equal to a named variable that will contain the Public ID for the entity being parsed, if it is available. If the Public ID is not available, an empty string is returned.

PARSER_URI

Set this keyword equal to a named variable that will contain the base URI (System ID) for the entity being parsed, if it is available. If the value is available, it is always an absolute URI. If the System ID is not available, an empty string is returned.

Note

Use this value to identify the document or external entity in diagnostics, or to resolve relative URIs.

IDLffXMLSAX::IgnorableWhitespace

The IDLffXMLSAX::IgnorableWhitespace procedure method is called when the parser detects whitespace that separates elements in an element content model.

Syntax

Obj `->` [IDLffXMLSAX::]IgnorableWhitespace, *Chars*

Arguments

Chars

A named variable that will contain the whitespace detected by the parser. Whitespace can consist of spaces, tabs, or newline characters in any combination.

Keywords

None.

IDLffXMLSAX::Init

The IDLffXMLSAX::Init function method initializes an XML parser object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. In most cases, you cannot call the Init method directly. However, one exception to this rule does exist. If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

Obj = OBJ_NEW('IDLffXMLSAX' [, /NAMESPACE_PREFIXES]
[, SCHEMA_CHECKING=[0,1,2] [, VALIDATION_MODE=[0,1,2]])

or

Result = *Obj* -> [IDLffXMLSAX::]Init( ) (*Only in a subclass' Init method*.)

Note

Keywords can be used in either form. They are omitted in the second form for brevity.

Return Value

Returns the object reference to this newly-created IDLffXMLSAX object.

Arguments

None.

Keywords

Properties you can retrieve via the IDLffXMLSAX::GetProperty are indicated by the word Get following the keyword. Properties you can set via the IDLffXMLSAX::SetProperty are indicated by the word Set following the keyword.

NAMESPACE_PREFIXES *(Get, Set)*

Set this keyword to indicate that namespace prefixes are enabled. By default, namespace prefixes are disabled.

SCHEMA_CHECKING *(Get, Set)*

XML *Schemas* describe the structure and allowed contents of an XML document. Schemas are more robust than, and are envisioned as a replacement for, DTDs. Set this keyword to an integer value to indicate the type of validation the parser should perform. By default, the parser will validate the parsed XML file against the specified schema, if one is provided; if no schema is provided, no validation will occur. Possible values are:

Value	Description
0	No validation.
1	Validate only if a schema is provided (the default).
2	Perform full schema constraint checking, if a schema is provided. This feature checks the schema grammar itself for additional errors. It does not affect the level of checking performed on document instances that use schema grammars.

Table 2-1: SCHEMA_CHECKING Values

VALIDATION_MODE *(Get, Set)*

XML *Document Type Definitions* (DTDs) describe the structure and allowed contents of an XML document. Set this keyword to indicate the type of XML validation that the parser should perform. By default, the parser will validate the parsed XML file against the specified DTD, if one is provided; if no DTD is provided, no validation will occur. Possible values are:

Value	Description
0	No validation.
1	Validate only if a DTD is provided (the default).
2	Always perform validation. If this option is in force and no DTD is provided, every XML element in the document will generate an error.

Table 2-2: VALIDATION_MODE Values

IDLffXMLSAX::InternalEntityDecl

The IDLffXMLSAX::InternalEntityDecl procedure method is called when the parser detects an `<!ENTITY ...>` declaration in a DTD for (parsed) internal entities. The entity can be either a general entity or a parameter entity.

Syntax

Obj `->` [IDLffXMLSAX::]InternalEntityDecl, *Name*, *Value*

Arguments

Name

A named variable that will contain the entity name. Names that start with the “%” character are parameter entities; all others are general entities.

Value

A named variable that will contain the entity value. The entity value can contain arbitrary XML content, which will be reparsed when the entity is expanded.

Keywords

None.

IDLffXMLSAX::NotationDecl

The IDLffXMLSAX::NotationDecl procedure method is called when the parser detects a `<!NOTATION ...>` declaration in a DTD.

Syntax

Obj `->` [IDLffXMLSAX::]NotationDecl, *Name*, *PublicID*, *SystemID*

Arguments

Name

A named variable that will contain the notation name.

PublicID

A named variable that will contain the Public ID for the notation.

Note

If this value is not specified in the notation declaration, this variable will contain an empty string.

SystemID

A named variable that will contain the System ID for the notation, provided as an absolute URI.

Note

If this value is not specified in the notation declaration, this variable will contain an empty string.

Keywords

None.

IDLffXMLSAX::ParseFile

The IDLffXMLSAX::ParseFile procedure method parses the specified XML file. During the parsing operation, different object methods are called as different items within the XML file are detected. When this method returns, the parse operation is complete.

Syntax

Obj `->` [IDLffXMLSAX::]ParseFile, *Filename*

Arguments

Filename

A string containing the full path name of the XML file to parse.

Keywords

None.

IDLffXMLSAX::ProcessingInstruction

The IDLffXMLSAX::ProcessingInstruction procedure method is called when the parser detects a processing instruction.

Syntax

Obj `->` [IDLffXMLSAX::]ProcessingInstruction, *Target*, *Data*

Arguments

Target

A named variable that will contain a string specifying the target, which is the application that should process the instruction.

Data

A named variable that will contain a string specifying the data to be passed to the application specified by *Target*.

Keywords

None.

IDLffXMLSAX::SetProperty

The IDLffXMLSAX::SetProperty procedure method is used to set the values of various properties of the parser.

Syntax

Obj `->` [IDLffXMLSAX::]SetProperty [, /NAMESPACE_PREFIXES] [, SCHEMA_CHECKING=[0,1,2] [, VALIDATION_MODE=[0,1,2]]

Arguments

None.

Keywords

Any keyword to the IDLffXMLSAX::Init followed by `Set` can be set using IDLffXMLSAX::SetProperty.

IDLffXMLSAX::SkippedEntity

The IDLffXMLSAX::SkippedEntity procedure method is called when the parser skips an entity and validation is not being performed. This method is rarely called by SAX parsers.

Syntax

Obj -> [IDLffXMLSAX::]SkippedEntity, *Name*

Arguments

Name

A named variable that will contain the name of the entity that was skipped.

Keywords

None.

IDLffXMLSAX::StartCDATA

The IDLffXMLSAX::StartCDATA procedure method is called when the parser detects the beginning of a `<[CDATA[...]]>` text section.

Syntax

Obj `->` [IDLffXMLSAX::]StartCDATA

Arguments

None.

Keywords

None.

IDLffXMLSAX::StartDocument

The IDLffXMLSAX::StartDocument procedure method is called when the parser begins processing a document, and before any data is processed.

Syntax

Obj `->` [IDLffXMLSAX::]StartDocument

Arguments

None.

Keywords

None.

IDLffXMLSAX::StartDTD

The IDLffXMLSAX::StartDTD procedure method is called when the parser detects the beginning of a Document Type Definition (DTD).

Syntax

Obj `->` [IDLffXMLSAX::]StartDTD, *Name*, *PublicID*, *SystemID*

Arguments

Name

A named variable that will contain the declared name of the root element for the document.

PublicID

A named variable that will contain the *normalized* version of the Public ID (a URI) declared for the external subset, or an empty string if no external subset was declared. *Normalization* involves removal of unnecessary "`.`" and "`..`" segments from the URI.

SystemID

A named variable that will contain the System ID (a URI) of the external subset, or an empty string if no external subset was declared.

Note

This URI has not been resolved into an absolute URI.

Keywords

None.

IDLffXMLSAX::StartElement

The IDLffXMLSAX::StartElement procedure method is called when the parser detects the beginning of an element.

Syntax

Obj `->` [IDLffXMLSAX::]StartElement, *URI*, *Local*, *qName* [, *attName*, *attValue*]

Arguments

URI

A named variable that will contain the namespace URI with which the element is associated, if any.

Local

A named variable that will contain the element name with any prefix removed, if the element is associated with a namespace URI. If the element is not associated with a namespace URI, this variable will contain an empty string.

qName

A named variable that will contain the element name found in the XML file.

Note

If the element is associated with a namespace URI, this variable may contain an empty string.

attrName

A named variable that will contain a string array, which is the names of the attributes associated with the element, if any.

attrValue

A named variable that will contain a string array, which is the values of each attribute associated with the element, if any. The returned array will have the same number of elements as the array returned in the *attrName* keyword variable.

Keywords

None.

IDLffXMLSAX::StartEntity

The IDLffXMLSAX::StartEntity procedure method is called when the parser detects the start of an internal or external entity expansion.

Syntax

Obj `->` [IDLffXMLSAX::]StartEntity, *Name*

Arguments

Name

A named variable that will contain the name of the entity.

Keywords

None.

IDLffXMLSAX::StartPrefixMapping

The IDLffXMLSAX::StartPrefixMapping procedure method is called when the parser detects the beginning of a namespace declaration.

Syntax

Obj -> [IDLffXMLSAX::]StartPrefixmapping, *Prefix*, *URI*

Arguments

Prefix

A named variable that will contain the prefix, which is being mapped. If the variable specified by *Prefix* contains an empty string, the mapping is for the default element namespace.

URI

A named variable that will contain the URI of the prefix namespace.

Keywords

None.

IDLffXMLSAX::StopParsing

Call the IDLffXMLSAX::StopParsing procedure method during a parse operation to halt the operation and cause the ParseFile method to return. This may be useful when parsing large XML files and the desired information is known to have been returned.

Syntax

Obj -> [IDLffXMLSAX::]StopParsing

Arguments

None.

Keywords

None.

IDLffXMLSAX::UnparsedEntityDecl

The IDLffXMLSAX::UnparsedEntityDecl procedure method is called when the parser detects an `<!ENTITY ...>` declaration that includes the NDATA keyword, indicating that the entity is not meant to be parsed. The value of the NDATA keyword generally specifies the name of a *notation*, which in turn specifies the type of data.

Syntax

Obj `->` [IDLffXMLSAX::]UnparsedEntityDecl, *Name*, *PublicID*, *SystemID*, *Notation*

Arguments

Name

A named variable that will contain the name of the unparsed entity.

PublicID

A named variable that will contain the Public ID of the notation specified by the entity’s NDATA keyword, or an empty string if no Public ID was declared.

SystemID

A named variable that will contain the System ID of the notation specified by the entity’s NDATA keyword. This value is normally an absolute URI.

Notation

A named variable containing the name of the notation specified by the entity’s NDATA keyword.

Keywords

None.

IDLffXMLSAX::Warning

The IDLffXMLSAX::Warning procedure method is called when the parser detects a problem during processing. This method prints an IDL error string to the IDL output log and allows the parser to continue processing.

Note

This method will cause error messages to be printed to the IDL output log. If you would like your application to hide error messages from the user (or display them in some other fashion), override this method in your subclass of the IDLffXMLSAX object class. If you do override this method, the error message will not be printed to the output log unless you explicitly call the superclass method.

Syntax

Obj `->` [IDLffXMLSAX::]Warning, *SystemID*, *LineNumber*, *ColumnNumber*, *Message*

Arguments

SystemID

A named variable that will contain the URI of the text that generated the error.

LineNumber

A named variable that contains the line number that contains the error.

ColumnNumber

A named variable that contains the column number that contains the error.

Message

A named variable that contains the error message.

Keywords

None.

IDLgrContour object

The following methods have been added in IDL 5.6:

- IDLgrContour::AdjustLabelOffsets
- IDLgrContour::GetLabelInfo

IDLgrContour::AdjustLabelOffsets

The IDLgrContour::AdjustLabelOffsets procedure method adjusts the offsets at which contour labels are positioned.

Syntax

Obj->[IDLgrContour::]AdjustLabelOffsets, *LevelIndex*, *LabelOffsets*

Arguments

LevelIndex

The index of the contour level for which the label offsets are being adjusted. This value must be greater than or equal to zero and less than the number of levels (refer to the N_LEVELS keyword in the IDLgrContour::Init method).

LabelOffsets

A scalar or vector of floating point offsets, [t0, t1, …], that indicate the parametric offsets along the length of each contour line at which each label is to be positioned. The number of elements in this vector must exactly match the number of elements returned in the LABEL_OFFSETS vector retrieved via the IDLgrContour::GetLabelInfo method for the same level.

Keywords

None.

IDLgrContour::GetLabelInfo

The IDLgrContour::GetLabelInfo procedure method retrieves information about the labels for a contour. The returned information is only valid until the next time the C_LABEL_INTERVAL or C_LABEL_OBJECTS property is modified using the IDLgrContour::SetProperty method, or the offsets are adjusted using the IDLgrContour::AdjustLabelOffsets method.

Syntax

Obj->[IDLgrContour::]GetLabelInfo, *Destination*, *LevelIndex*
[, LABEL_OFFSETS=*variable*] [, LABEL_POLYS=*variable*]
[, LABEL_OBJECTS=*variable*]

Arguments

Destination

A reference to a destination object (such as an IDLgrWindow or IDLgrBuffer object). The contour label information will be computed so that the requested font size is satisfied for this destination device.

LevelIndex

The index of the contour level for which the label information is being requested. This value must be greater than or equal to zero and less than the number of levels (refer to the N_LEVELS keyword in the IDLgrContour::Init method).

Keywords

LABEL_OFFSETS

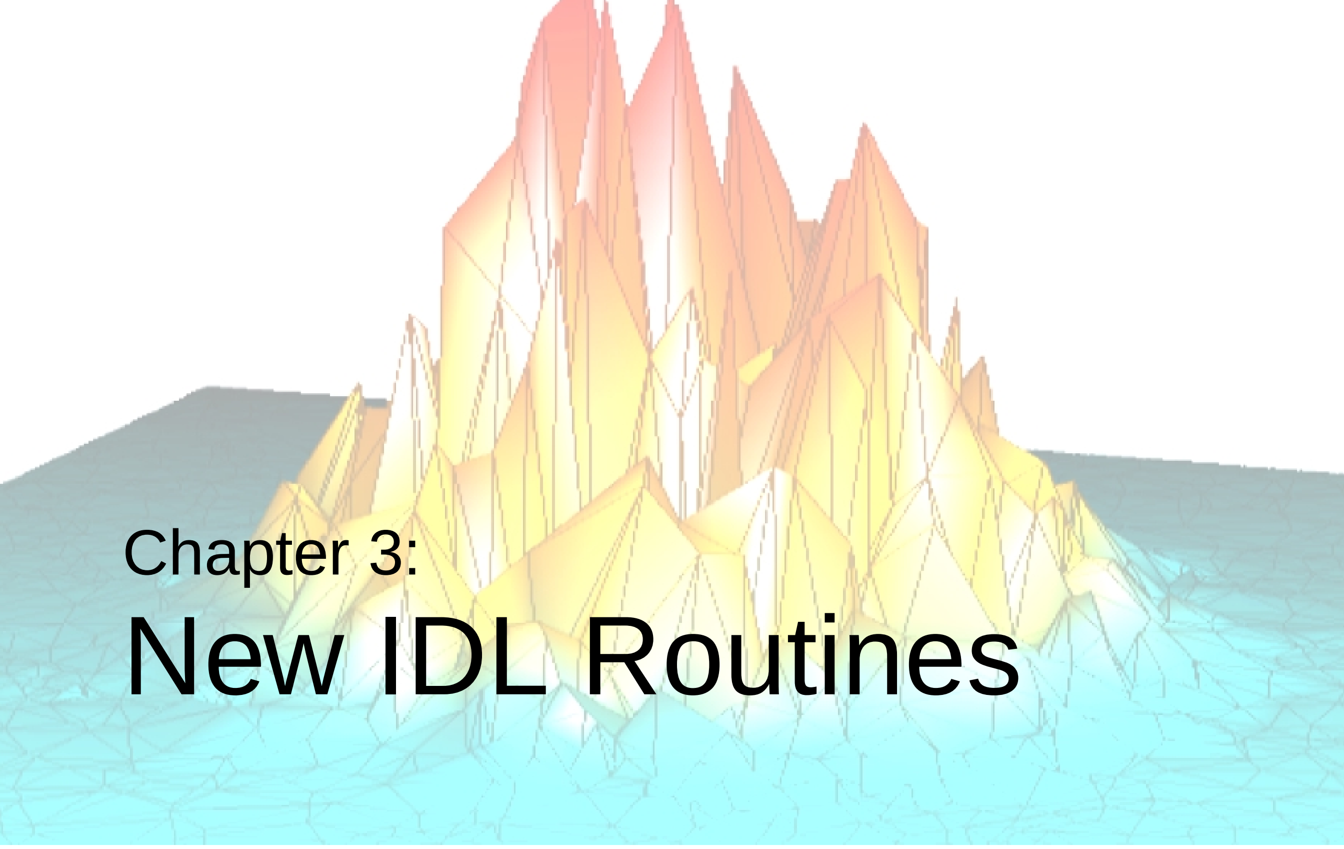
Set this keyword to a named variable that upon return will contain a vector of floating point offsets, [t0, t1, …], that indicate the parametric offsets along the length of each contour line at which the contour labels are positioned.

LABEL_POLYLINES

Set this keyword to a named variable that upon return will contain a vector of contour polyline indices, [$P_{0,}$ P_1, …], that indicate which contour lines are labeled. P_i corresponds to the ith contour line. Note that if a given contour line has more than one label along its perimeter, then the corresponding polyline index may appear more than once in the LABEL_POLYLINES vector.

LABEL_OBJECTS

Set this keyword to a named variable that upon return will contain a vector of objects that represent the labels for each contour label.



Chapter 3: New IDL Routines

This chapter describes routines introduced in IDL version 5.6

COPY_LUN

The COPY_LUN procedure copies data between two open files. It allows you to transfer a known amount of data from one file to another without needing to have the data available in an IDL variable. COPY_LUN can copy a fixed amount of data, specified in bytes or lines of text, or it can copy from the current position of the file pointer in the input file to the end of that file.

COPY_LUN copies data between open files. To copy entire files based on their names, see the FILE_COPY procedure. To read and discard a known amount of data from a file, see the SKIP_LUN.

Syntax

COPY_LUN, *FromUnit*, *ToUnit* [, *Num*] [, /EOF] [, /LINES]
[, /TRANSFER_COUNT]

Arguments

FromUnit

An integer that specifies the file unit for the file from which data is to be taken (the *source* file). Data is copied from *FromUnit*, starting at the current position of the file pointer. The file pointer is advanced as data is read. The file specified by *FromUnit* must be open, and must not have been opened with the RAWIO keyword to OPEN.

ToUnit

An integer that specifies the file unit for the file to which data is to be written (the *destination* file). Data is written to *ToUnit*, starting at the current position of the file pointer. The file pointer is advanced as data is written. The file specified by *ToUnit* must be open for output (OPENW or OPENU), and must not have been opened with the RAWIO keyword to OPEN.

Num

The amount of data to transfer between the two files. This value is specified in bytes, unless the LINES keyword is specified, in which case it is taken to be the number of text lines. If *Num* is not specified, COPY_LUN acts as if the EOF keyword has been set, and copies all data in *FromUnit* (the source file) from the current position of the file pointer to the end of the file.

If *Num* is specified and the source file comes to end of file before the specified amount of data is transferred, COPY_LUN issues an end-of-file error. The EOF keyword alters this behavior.

Keywords

EOF

Set this keyword to ignore the value given by *Num* (if present) and instead transfer all data between the current position of the file pointer in *FromUnit* and the end of the file.

Note

If EOF is set, no end-of-file error is issued even if the amount of data transferred does not match the amount specified by *Num*. The TRANSFER_COUNT keyword can be used with EOF to determine how much data was transferred.

LINES

Set this keyword to indicate that the *Num* argument specifies the number of lines of text to be transferred. By default, the *Num* argument specifies the number of bytes of data to transfer.

TRANSFER_COUNT

Set this keyword equal to a named variable that will contain the amount of data transferred. If LINES is specified, this value is the number of lines of text. Otherwise, it is the number of bytes. TRANSFER_COUNT is primarily useful when the *Num* argument is not specified or the EOF keyword is present. If *Num* is specified and the EOF keyword is not present, TRANSFER_COUNT will be the same as the value specified for *Num*.

Examples

Copy the next 100000 bytes of data between two files:

```
COPY_LUN, FromUnit, ToUnit, 100000
```

Copy the next 8 lines of text between two files:

```
COPY_LUN, FromUnit, ToUnit, 8, /LINES
```

Copy the remainder of the data in one file to another, and use the TRANSFER_COUNT keyword to determine how much data was copied:

```
COPY_LUN, FromUnit, ToUnit, /EOF, TRANSFER_COUNT=n
```

Copy the remaining lines of text from one file to another, and use the TRANSFER_COUNT keyword to determine how many lines were transferred.

```
COPY_LUN, FromUnit, ToUnit, /EOF, /LINES, TRANSFER_COUNT=n
```

Version History

Introduced: 5.6

See Also

CLOSE, EOF, FILE_COPY, FILE_LINK, FILE_MOVE, OPEN, READ/READF, SKIP_LUN, WRITEU

DIAG_MATRIX

The DIAG_MATRIX function constructs a diagonal matrix from an input vector, or if given a matrix, then DIAG_MATRIX will extract a diagonal vector.

Syntax

Result = DIAG_MATRIX(*A* [, *Diag*])

Return Value

- If given an input vector with *n* values, the result is an *n*-by-*n* array of the same type. The DIAG_MATRIX function may also be used to construct subdiagonal or superdiagonal arrays.
- If given an input *n*-by-*m* array, the result is a vector with MIN(*n*,*m*) elements containing the diagonal elements. The DIAG_MATRIX function may also be used to extract subdiagonals or superdiagonals.

Arguments

A

Either an *n*-element input vector to convert to a diagonal matrix, or a *n*-by-*m* input array to extract a diagonal. *A* may be any numeric type.

Diag

An optional argument that specifies the subdiagonal (*Diag* < 0) or superdiagonal (*Diag* > 0) to fill or extract. The default is *Diag*=0 which puts or extracts the values along the diagonal. If *A* is a vector with the *m* elements, then the result is an *n*-by-*n* array, where *n* = *m* + *ABS*(*Diag*). If *A* is an array, then the result is a vector whose length depends upon the number of elements remaining along the subdiagonal or superdiagonal.

Tip

The *Diag* argument may be used to easily construct tridiagonal arrays. For example, the expression,

```
DIAG_MATRIX(VL,-1) + DIAG_MATRIX(V) + DIAG_MATRIX(VU,1)
```

will create an *n*-by-*n* array, where *VL* is an (*n* - 1)-element vector containing the subdiagonal values, *V* is an *n*-element vector containing the diagonal values, and *VU* is an (*n* - 1)-element vector containing the superdiagonal values.

Keywords

None.

Example

Create a tridiagonal matrix and extract the diagonal using the following program:

```
PRO ExDiagMatrix
; Convert three input vectors to a tridiagonal matrix:
diag = [1, -2, 3, -4]
sub = [5, 10, 15]
super = [3, 6, 9]
array = DIAG_MATRIX(diag) + $
DIAG_MATRIX(super, 1) + DIAG_MATRIX(sub, -1)
PRINT, 'DIAG_MATRIX array:'
PRINT, array

; Extract the diagonal:
PRINT, 'DIAG_MATRIX diagonal:'
PRINT, DIAG_MATRIX(array)
END
```

When this program is compiled and run, IDL prints:

```
DIAG_MATRIX array:
1       3       0       0
5      -2       6       0
0      10       3       9
0       0      15      -4
DIAG_MATRIX diagonal:
1      -2       3      -4
```

Version History

Introduced: 5.6

See Also

IDENTITY, MATRIX_MULTIPLY, MATRIX_POWER, “Multiplying Arrays” in Chapter 22 of the *Using IDL* manual.

FILE_COPY

The FILE_COPY procedure copies files, or directories of files, to a new location. The copies retain the permission settings of the original files, and belong to the user that performed the copy. See "Rules Used By FILE_COPY" on page 198 for additional information.

FILE_COPY copies files based on their names. To copy data between open files, see the COPY_LUN procedure.

Syntax

FILE_COPY, *SourcePath*, *DestPath* [, /ALLOW_SAME] [, /NOEXPAND_PATH] [, /OVERWRITE] [, /RECURSIVE] [, /REQUIRE_DIRECTORY] [, /VERBOSE]

UNIX-Only Keywords: [, /COPY_NAMED_PIPE] [, /COPY_SYMLINK] [, /FORCE]

Arguments

SourcePath

A scalar string or string array containing the names of the files or directories to be copied.

Note

If *SourcePath* contains a directory, the RECURSIVE keyword must be set.

DestPath

A scalar string or string array containing the names of the destinations to which the files and directories specified by *SourcePath* are to be copied. If more than one file is to be copied to a given destination, that destination must exist and be a directory.

Keywords

ALLOW_SAME

Attempting to copy a file on top of itself by specifying the same file for *SourcePath* and *DestPath* is usually considered to be an error. If the ALLOW_SAME keyword is set, no copying is done and the operation is considered successful.

COPY_NAMED_PIPE (UNIX Only)

When FILE_COPY encounters a UNIX *named pipe* (also called a *fifo*) in *SourcePath*, it usually opens it as a regular file and attempts to copy data from it to the destination file. If COPY_NAMED_PIPE is set, FILE_COPY will instead replicate the pipe, creating a new named pipe at the destination using the system `mkfifo()` function.

COPY_SYMLINK (UNIX Only)

When FILE_COPY encounters a UNIX *symbolic link* in *SourcePath*, it attempts to copy the file or directory pointed to by the link. If COPY_SYMLINK is set, FILE_COPY will instead create a symbolic link at the destination with the same name as the source symbolic link, and pointing to the same path as the source.

FORCE (UNIX Only)

Even if the OVERWRITE keyword is set, FILE_COPY does not overwrite files that have their file permissions set to prevent it. If the FORCE keyword is set, such files are quietly removed to make way for the overwrite operation to succeed.

Note

FORCE does not imply OVERWRITE; both must be specified to overwrite a protected file.

NOEXPAND_PATH

Set this keyword to cause FILE_COPY to use *SourcePath* and *DestPath* exactly as specified, without expanding any wildcard characters or environment variable names included in the paths. See FILE_SEARCH for details on path expansion.

OVERWRITE

Set this keyword to allow FILE_COPY to overwrite an existing file.

RECURSIVE

Set this keyword to cause directories specified by *SourcePath* to be copied to *DestPath* recursively, preserving the hierarchy and names of the files from the source. If *SourcePath* includes one or more directories, the RECURSIVE keyword *must* be set.

Note

On a UNIX system, when performing a recursive copy on a directory hierarchy that includes files that are links to other files, the destination files will be copies, not links. Setting the COPY_SYMLINK keyword will cause files that are *symbolic* links to be copied as symbolic links, but FILE_COPY does not include a similar facility for copying *hard* links. See the description of the FILE_LINK for more information on UNIX file links.

REQUIRE_DIRECTORY

Set this keyword to cause FILE_COPY to require that *DestPath* exist and be a directory.

VERBOSE

Set this keyword to cause FILE_COPY to issue an informative message for every file copy operation it carries out.

Rules Used By FILE_COPY

The following rules govern how FILE_COPY operates:

- The arguments to FILE_COPY can be scalar or array. If both arguments are arrays, the arrays must contain the same number of elements; in this case, the files are copied pairwise, with each file from *SourcePath* being copied to the corresponding file in the *DestPath*. If *SourcePath* is an array and *DestPath* is a scalar, all files in *SourcePath* are copied to the single location given by *DestPath*, which must exist and be a directory.
- Elements of *SourcePath* may use wildcard characters (as accepted by the FILE_SEARCH function) to specify multiple files. All the files matched for a given element of *SourcePath* are copied to the location specified by the corresponding element of *DestPath*. If multiple files are copied to a single element of *DestPath*, that element must exist and be a directory.
- If a file specified in *DestPath* does not exist, the corresponding file from *SourcePath* is copied using the name specified by *DestPath*. Any parent directories to the file specified by *DestPath* must already exist.
- If *DestPath* names an existing regular file, FILE_COPY will not overwrite it, unless the OVERWRITE keyword is specified.

- If *DestPath* names an existing directory and *SourcePath* names a regular (non-directory) file, then FILE_COPY creates a file with the same name as the file given by *SourcePath* within the *DestPath* directory.
- If *DestPath* specifies an existing directory and *SourcePath* also names a directory, and the RECURSIVE keyword is set, FILE_COPY checks for the existence of a subdirectory of *DestPath* with the same name as the source directory. If this subdirectory does not exist, it is created using the same permissions as the directory being copied. Then, all the files and directories underneath the source directory are copied to this subdirectory. FILE_COPY will refuse to overwrite existing files within the destination subdirectory unless the OVERWRITE keyword is in effect.

Examples

Make a backup copy of a file named `myroutine.pro` in the current working directory:

```
FILE_COPY, 'myroutine.pro', 'myroutine.pro.backup'
```

Create a subdirectory named BACKUP in the current working directory and copy all `.pro` files, `makefile`, and `mydata.dat` into it:

```
FILE_MKDIR, 'BACKUP'
FILE_COPY, ['*.pro', 'makefile', 'mydata.dat'], 'BACKUP'
```

Version History

Introduced: 5.6

See Also

COPY_LUN, FILE_LINK, FILE_MOVE

FILE_LINES

The FILE_LINES function reports the number of lines of text contained within the specified file or files.

Text files containing data are very common. To read such a file usually requires knowing how many lines of text it contains. Under UNIX and Windows, there is no special text file type, and it is not possible to tell how many lines are contained in a file from basic file attributes. Rather, lines are encoded using a special character or characters at the end of each line:

- UNIX operating systems use an ASCII linefeed (LF) character at the end of each line.
- Older Macintosh systems (prior to the UNIX-based Mac OS X) use a carriage return (CR).
- Microsoft Windows uses a two character CR/LF sequence.

The only way to determine the number of lines of text contained within a file is to open it and count lines while reading and skipping over them until the end of the file is encountered. Since files are often copied from one type of system to another without going through the proper line termination conversion, portable software needs to be able to recognize any of these terminations, regardless of the system being used. FILE_LINES performs this operation in an efficient and portable manner, handling all three of the line termination conventions listed above.

This routine works by opening the file and reading the data contained within. It is therefore only suitable for regular disk files, and only when access to that file is fast enough to justify reading it more than once. For other types of files, other approaches are necessary, such as:

- Reading the file once, using an adaptive (expandable) data structure, counting the number of lines as they are input, and growing the data structure as necessary.
- Building a header into your file format that includes the necessary information, or somehow embedding the number of lines into the file data.
- Maintaining file information in a separate file associated with each file.
- Using a self describing data format that avoids these issues.

This routine assumes that the specified file or files contain only lines of text. It is unable to correctly count lines in files that contain binary data, or which do not use the standard line termination characters. Results are undefined for such files.

Note that FILE_LINES is equivalent to the following IDL code:

```
FUNCTION file_lines, filename
   OPENR, unit, filename, /GET_LUN
   str = ''
   count = 0ll
   WHILE NOT EOF(unit) DO BEGIN
      READF, unit, str
      count = count + 1
   ENDWHILE
   FREE_LUN, unit
   RETURN, count
END
```

The primary advantage of FILE_LINES over the IDL version shown here is efficiency. FILE_LINES is able to avoid the overhead of the WHILE loop as well as not having to create an IDL string for each line of the file.

Syntax

Result = FILE_LINES(*Path* [, /NOEXPAND_PATH])

Return Value

Returns the number of lines of text contained within the specified file or files. If an array of file names is specified via the *Path* parameter, the return value is an array with the same number of elements as *Path*, with each element containing the number of lines in the corresponding file.

Arguments

Path

A scalar string or string array containing the names of the text files for which the number of lines is desired.

Keywords

NOEXPAND_PATH

If specified, FILE_LINES uses *Path* exactly as specified, without expanding any wildcard characters or environment variable names included in the path. See FILE_SEARCH for details on path expansion.

Examples

Read the contents of the text file `mydata.dat` into a string array.

```
nlines = FILE_LINES('mydata.dat')
sarr = STRARR(nlines)
OPENR, unit, 'mydata.dat',/GET_LUN
READF, unit, sarr
FREE_LUN, unit
```

Version History

Introduced: 5.6

See Also

READ/READF

FILE_LINK

The FILE_LINK procedure creates UNIX file links, both regular (hard) and symbolic. FILE_LINK is available only under UNIX.

A hard link is a directory entry that references a file. UNIX allows multiple such links to exist simultaneously, meaning that a given file can be referenced by multiple names. All such links are fully equivalent references to the same file (there are no concepts of primary and secondary names). All files carry a reference count that contains the number of hard links that point to them; deleting a link to a file does not remove the actual file from the filesystem until the last hard link to the file is removed. The following limitations on hard links are enforced by the operating system:

- Hard links may not span filesystems, as hard linking is only possible within a single filesystem.
- Hard links may not be created between directories, as doing so has the potential to create infinite circular loops within the hierarchical UNIX filesystem. Such loops will confuse many system utilities, and can even cause filesystem damage.

A symbolic link is an indirect pointer to a file; its directory entry contains the name of the file to which it is linked. Symbolic links may span filesystems and may refer to directories.

Many users find symbolic links easier to understand and use. Due to their generality and lack of restriction, RSI recommends their use over hard links for most purposes. FILE_LINK creates symbolic links by default.

See “Rules Used by FILE_LINK” on page 204 for information on how FILE_LINK interprets its arguments.

Syntax

FILE_LINK, *SourcePath*, *DestPath* [, /ALLOW_SAME] [, /HARDLINK] [, /NOEXPAND_PATH] [, /VERBOSE]

Arguments

SourcePath

A scalar string or string array containing the names of the files or directories to be linked.

DestPath

A scalar string or string array containing the names of the destinations to which the files and directories given by *SourcePath* are to be linked. If more than one file is to be linked to a given destination, that destination must exist and be a directory.

Keywords

ALLOW_SAME

Attempting to link a file to itself by specifying the same file for *SourcePath* and *DestPath* is usually considered to be an error. If the ALLOW_SAME keyword is set, no link is created and the operation is considered to be successful.

HARDLINK

Set this keyword to create hard links. By default, FILE_LINK creates symbolic links.

NOEXPAND_PATH

Set this keyword to cause FILE_LINK to use *SourcePath* and *DestPath* exactly as specified, without expanding any wildcard characters or environment variable names included in the paths. See FILE_SEARCH for details on path expansion.

VERBOSE

Set this keyword to cause FILE_LINK to issue an informative message for every file link operation it carries out.

Rules Used by FILE_LINK

The following rules govern how FILE_LINK operates:

- The arguments to FILE_LINK can be scalar or array. If both arguments are arrays, they must contain the same number of elements, and the files are paired, with each file from *SourcePath* being linked to the corresponding file in the *DestPath*. If *SourcePath* is an array and *DestPath* is a scalar, all links are created in the single location given by *DestPath*, which must exist and be a directory.
- Elements of *SourcePath* may use wildcard characters (as accepted by the FILE_SEARCH function) to specify multiple files. All the files matched for a given element of *SourcePath* are linked to the corresponding element of

DestPath. If multiple files are linked to a single element of *DestPath*, that element must exist and be a directory.

- If a file specified in *DestPath* does not exist, the corresponding file from *SourcePath* is linked using the name specified by *DestPath*. Any parent directories to the filename specified by *DestPath* must already exist.
- If *DestPath* names an existing regular file, FILE_LINK will not overwrite it.
- If *DestPath* names an existing directory, a link with the same name as the source file is created in the directory. This is primarily of interest with hard links.

Examples

Create a symbolic link named `current.dat` in the current working directory, pointing to the file `/master/data/saturn7.dat`:

```
FILE_LINK, '/master/data/saturn7.dat', 'current.dat'
```

Version History

Introduced: 5.6

See Also

COPY_LUN, FILE_COPY, FILE_MOVE, FILE_READLINK

FILE_MOVE

The FILE_MOVE procedure renames files and directories, effectively moving them to a new location. The moved files retain their permission and ownership attributes. Within a given filesystem or volume, FILE_MOVE does not copy file data. Rather, it simply changes the file names by updating the directory structure of the filesystem. This operation is fast and safe, but is only possible within a single filesystem. Attempts to move a regular file from one filesystem to another are carried out by copying the file using FILE_COPY, and then deleting the original file. It is an error to attempt to use FILE_MOVE to move a directory from one filesystem to another.

See "Rules Used by FILE_MOVE" on page 207 for information on how FILE_MOVE interprets its arguments.

Syntax

FILE_MOVE, *SourcePath*, *DestPath* [, /ALLOW_SAME] [, /NOEXPAND_PATH] [, /OVERWRITE] [, /REQUIRE_DIRECTORY] [, /VERBOSE]

Arguments

SourcePath

A scalar string or string array containing the names of the files or directories to be moved.

DestPath

A scalar string or string array containing the names of the destinations to which the files and directories specified by *SourcePath* are to be moved. If more than one file is to be moved to a given destination, that destination must exist and be a directory.

Keywords

ALLOW_SAME

Attempting to move a file on top of itself by specifying the same file for *SourcePath* and *DestPath* is usually considered to be an error. If the ALLOW_SAME keyword is set, no renaming is done and the operation is considered to be successful.

NOEXPAND_PATH

Set this keyword to cause FILE_MOVE to use *SourcePath* and *DestPath* exactly as specified, without expanding any wildcard characters or environment variable names included in the paths. See FILE_SEARCH for details on path expansion.

OVERWRITE

Set this keyword to allow FILE_MOVE to overwrite an existing file.

REQUIRE_DIRECTORY

Set this keyword to cause FILE_MOVE to require that *DestPath* exist and be a directory.

VERBOSE

Set this keyword to cause FILE_MOVE to issue an informative message for every file move operation it carries out.

Rules Used by FILE_MOVE

The following rules govern how FILE_MOVE operates:

- The arguments to FILE_MOVE can be scalar or array. If both arguments are arrays, they must contain the same number of elements, and the files are moved in pairs, with each file from *SourcePath* being renamed to the corresponding file in the *DestPath*. If *SourcePath* is an array and *DestPath* is a scalar, all files in *SourcePath* are renamed to the single location given by *DestPath*, which must exist and be a directory.
- Elements of *SourcePath* may use wildcard characters (as accepted by the FILE_SEARCH function) to specify multiple files. All the files matched for that element of *SourcePath* are renamed to the location specified by the corresponding element of *DestPath*. If multiple files are renamed to a single element of *DestPath*, that element must exist and be a directory.
- If a file specified in *DestPath* does not exist, the corresponding file from *SourcePath* is moved using the name specified by *DestPath*. Any parent directories to the filename specified by *DestPath* must already exist.
- If *DestPath* names an existing regular file, FILE_MOVE will not overwrite it, unless the OVERWRITE keyword is specified.
- If *DestPath* names an existing directory and *SourcePath* names a regular (non-directory) file, the source file is moved into the specified directory.

- If *DestPath* specifies an existing directory and *SourcePath* also names a directory, FILE_MOVE checks for the existence of a subdirectory of *DestPath* with the same name as the source directory. If this subdirectory does not exist, the source directory is moved to the specified location. If the subdirectory does exist, an error is issued, and the rename operation is not carried out.

Examples

Rename the file `backup.dat` to `primary.dat` in the current working directory:

```
FILE_MOVE, 'backup.dat', 'primary.dat'
```

Create a subdirectory named `BACKUP` in the current working directory and move all `.pro` files, `makefile`, and `mydata.dat` into it:

```
FILE_MKDIR, 'BACKUP'
FILE_MOVE, ['*.pro', 'makefile', 'mydata.dat'], 'BACKUP'
```

Version History

Introduced: 5.6

See Also

COPY_LUN, FILE_COPY, FILE_LINK

FILE_READLINK

The FILE_READLINK function returns the path pointed to by UNIX symbolic links.

Syntax

Result = FILE_READLINK(*Path* [, /ALLOW_NONEXISTENT]
[, /ALLOW_NONSYMLINK] [, /NOEXPAND_PATH])

Return Value

Returns the path associated with a symbolic link.

Arguments

Path

A scalar string or string array containing the names of the symbolic links to be translated.

Keywords

ALLOW_NONEXISTENT

Set this keyword to return a NULL string rather than throwing an error if *Path* contains a non-existent file.

ALLOW_NONSYMLINK

Set this keyword to return a NULL string rather than throwing an error if *Path* contains a path to a file that is not a symbolic link.

NOEXPAND_PATH

Set this keyword to cause FILE_READLINK to use *Path* exactly as specified, without expanding any wildcard characters or environment variable names included in the path. See FILE_SEARCH for details on path expansion.

Examples

Under Mac OS X, the `/etc` directory is actually a symbolic link. The following statement reads it and returns the location to which the link points:

```
path = FILE_READLINK('/etc')
```

It is possible to have chains of symbolic links, each pointing to another. The following function uses FILE_READLINK to iteratively translate such links until it finds the actual file:

```
FUNCTION RESOLVE_SYMLINK, path

  savepath = path      ; Remember last successful translation
  WHILE (path NE '') DO BEGIN
    path = FILE_READLINK(path, /ALLOW_NONEXISTENT, $
      /ALLOW_NONSYMLINK)
    IF (path NE '') THEN BEGIN
      ; If returned path is not absolute, use it to replace the
      ; last path segment of the previous path.
      IF (STRMID(path, 0, 1) NE '/') THEN BEGIN
        last = STRPOS(savepath, '/', /REVERSE_SEARCH)
        IF (last NE -1) THEN path = STRMID(savepath, 0, last) $
          + '/' + path
      ENDIF
      savepath = path
    ENDIF
  ENDWHILE

  ; FILE_EXPAND_PATH removes redundant things like /./ from
  ; the result.
  RETURN, FILE_EXPAND_PATH(savepath)

END
```

Version History

Introduced: 5.6

See Also

FILE_LINK

FILE_SAME

It is common for a given file to be accessible via more than one name. For example, a relative path and a fully-qualified path to the same file are considered different names, since the strings that make up the paths are not lexically identical. In addition, under UNIX, the widespread use of links (hard and symbolic) makes multiple names for the same file very common.

The FILE_SAME function is used to determine if two different file names refer to the same underlying file.

The mechanism used to determine whether two names refer to the same file depends on the operating system in use:

UNIX: Under UNIX, all files are uniquely identified by two integer values: the filesystem that contains the file and the *inode number*, which identifies the file within the filesystem. If the input arguments are lexically identical, FILE_SAME will return True, regardless of whether the file specified actually exists. Otherwise, FILE_SAME compares the device and inode numbers of the two files, and returns True if they are identical, or False otherwise.

Windows: Unlike UNIX, Microsoft Windows identifies files solely by their names. FILE_SAME therefore expands the two supplied paths to their fully qualified forms, and then performs a simple case insensitive string comparison to determine if the paths are identical. This is reliable for local disk files, but can produce incorrect results under some circumstances:

- UNC network paths can expand to different, but equivalent, paths. For example, a network server may be referred to by either a name or an IP address.
- Network attached storage can have mechanisms for giving multiple names to the same file, but to the Windows client system the names will appear to refer to different files. For example, a UNIX server using Samba software to serve files to machines on a Windows network can use symbolic links to produce two names for the same file, but these will appear as two distinct files to a Windows machine.

For these reasons, FILE_SAME is primarily of interest on UNIX systems. Under Windows, RSI recommends its use only on local files.

Syntax

Result = FILE_SAME(*Path1*, *Path2* [, /NOEXPAND_PATH])

Return Value

FILE_SAME returns True (1) if two filenames refer to the same underlying file, or False (0) otherwise. If either or both of the input arguments are arrays of file names, the result is an array, following the same rules as standard IDL operators.

Arguments

Path1, Path2

Scalar or array string values containing the two file paths to be compared.

Keywords

NOEXPAND_PATH

Set this keyword to cause FILE_SAME to use the *Path* arguments exactly as specified, without expanding any wildcard characters or environment variable names included in the paths. See FILE_SEARCH for details on path expansion. The utility of doing this depends on the operating system in use:

UNIX: Under UNIX, path expansion is not necessary unless the *Path* arguments use shell meta characters or environment variables.

Windows: By default, FILE_SAME expands the supplied paths to their fully qualified forms in order to be able to compare them. Preventing this path expansion cripples its ability to make a useful comparison, and is not recommended.

Examples

UNIX command shells often provide the `HOME` environment variable to point at the user's home directory. Many shells also expand the `'~'` character to point at the home directory. The following IDL statement determines if these two mechanisms refer to the same directory:

```
PRINT, FILE_SAME('~', '$HOME')
```

On a UNIX system, the following statement determines if the current working directory is the same as your home directory:

```
PRINT, FILE_SAME('.', '$HOME')
```

On some BSD-derived UNIX systems, the three commands `/bin/cp`, `/bin/ln`, and `/bin/mv` are actually three hard links to the same binary file. The following statement will print the number 1 if this is true on your system:

```
PRINT, TOTAL(FILE_SAME('/bin/cp', ['/bin/ln', '/bin/mv'])) EQ 2
```

Under Mac OS X, the `/etc` directory is actually a symbolic link to `/private/etc`. As a result, the following lines of code provide a simple test to determine whether Mac OS X is the current platform:

```
IF FILE_SAME('/etc', '/private/etc') THEN $
   PRINT, 'Running Mac OS X' ELSE $
   PRINT, 'Not Running Mac OS X'
```

Note

The above lines are shown simply as an example; checking the value of !VERSION.OS_FAMILY is a more reliable method of determining which operating system is in use.

Version History

Introduced: 5.6

See Also

FILE_EXPAND_PATH, FILE_INFO, FILE_SEARCH, FILE_TEST

H5_BROWSER

The H5_BROWSER function presents a graphical user interface for viewing and reading HDF5 files. The browser provides a tree view of the HDF5 file or files, a data preview window, and an information window for the selected objects. The browser may be created as either a selection dialog with Open/Cancel buttons, or as a standalone browser that can import data to the IDL main program level.

Note

This function is not part of the standard HDF5 interface, but is provided as a programming convenience.

Syntax

Result = H5_BROWSER([*Files*] [, /DIALOG_READ])

Return Value

If the DIALOG_READ keyword is specified then the *Result* is a structure containing the selected group or dataset (as described in the H5_PARSE function), or a zero if the Cancel button was pressed. If the DIALOG_READ keyword is not specified then the *Result* is the widget ID of the HDF5 browser.

Arguments

Files

An optional scalar string or string array giving the name of the files to initially open. Additional files may be opened interactively. If *Files* is not provided then the user is automatically presented with a File Open dialog upon startup.

Keywords

DIALOG_READ

If this keyword is set then the HDF5 browser is created as a modal Open/Cancel dialog instead of a standalone GUI. In this case, the IDL command line is blocked, and no further input is taken until the Open or Cancel button is pressed. If the GROUP_LEADER keyword is specified, then that widget ID is used as the group leader, otherwise a default group leader base is created.

All keywords to WIDGET_BASE, such as GROUP_LEADER and TITLE, are passed on to the top-level base.

Graphical User Interface Options

Open HDF5 file

Click on this button to bring up a file selection dialog. Multiple files may be selected for parsing. All selected files are added to the tree view.

Show preview

If this toggle button is selected, then the data within datasets will be shown in the preview window. One-dimensional datasets will be shown as line plots. Two-dimensional datasets will be shown as images, along with any provided image palettes. For three or higher-dimensional datasets, a two dimensional slice will be shown.

Fit in window

If this toggle button is selected, then the preview image will be scaled larger or smaller to fit within the preview window. The aspect ratio of the image will be unchanged.

Flip vertical

If this toggle button is selected, then the preview image will flipped from top to bottom.

Flip horizontal

If this toggle button is selected, then the preview image will flipped from left to right.

Note

If the DIALOG_READ keyword is present then the following options are available:

Open

Click on this button to close the HDF5 browser, and return an IDL structure containing the selected group or dataset, as described in the H5_PARSE function.

Cancel

Click on this button to close the HDF5 browser, and return a scalar zero for the result.

Note

If the DIALOG_READ keyword is not present then the following options are available:

Variable name for import

Set this text string to the name of the IDL variable to construct when importing HDF5 data to IDL structures. If the entered name is not a valid IDL identifier, then a valid identifier will be constructed by converting all non-alphanumeric characters to underscores.

Include data

If this toggle button is selected, then all data within the selected datasets will be read in from the HDF5 file and included in the IDL structure.

Import to IDL

Click on this button to import the currently selected HDF5 object into the IDL main program level. Imported variables will consist of a nested hierarchy of IDL structures, as described in the H5_PARSE function.

Done

Click on this button to close the HDF5 browser.

Example

The following example starts up the HDF5 browser on a sample file:

```
File = FILEPATH('hdf5_test.h5', SUBDIR=['examples','data'])
Result = H5_BROWSER(File)
```

Version History

Introduced 5.6

See Also

H5_PARSE

H5_CLOSE

The H5_CLOSE procedure flushes all data to disk, closes file identifiers, and cleans up memory. This routine closes IDL's link to its HDF5 libraries. This procedure is used automatically by IDL when RESET_SESSION is issued, but it may also be called if the user desires to free all HDF5 resources.

Syntax

H5_CLOSE

Arguments

None.

Keywords

None.

Version History

Introduced 5.6

See Also

H5_OPEN

H5_GET_LIBVERSION

The H5_GET_LIBVERSION function returns the current version of the HDF5 library used by IDL.

Syntax

Result = H5_GET_LIBVERSION( )

Return Value

Returns a string in the form of '*maj.min.rel*', where *maj* is the major number, *min* is the minor number, and *rel* is the release number. An example would be '1.4.3', representing HDF5 version 1.4.3.

Arguments

None.

Keywords

None.

Version History

Introduced 5.6

See Also

H5_OPEN

H5_OPEN

The H5_OPEN procedure initializes IDL's HDF5 library. This procedure is issued automatically by IDL when one of IDL's HDF5 routines is used.

Note

This routine is provided for diagnostic purposes only. You do not need to use this routine while working with IDL's HDF5 routines.

Syntax

H5_OPEN

Arguments

None.

Keywords

None.

Version History

Introduced 5.6

See Also

H5_CLOSE, H5_GET_LIBVERSION

H5_PARSE

The H5_PARSE function recursively descends through an HDF5 file or group and creates an IDL structure containing object information and data.

Note

This function is not part of the standard HDF5 interface, but is provided as a programming convenience.

Syntax

Result = H5_PARSE (*File* [, /READ_DATA])

or

Result = H5_PARSE (*Loc_id*, *Name* [, FILE=*string*] [, PATH=*string*] [, /READ_DATA])

Return Value

The *Result* is an IDL structure containing the parsed file or group. The fields within each structure in *Result* depend upon the object type.

Structure Fields Common to All Object Types

Field	Description
_NAME	Object name, or the filename if at the top level
_ICONTYPE	Name of associated icon, used by H5_BROWSER
_TYPE	Object type, such as GROUP, DATASET, DATATYPE, ATTRIBUTE, or LINK

Table 3-1: Structure Fields Common to All Object Types

Additional Fields for Groups, Datasets, and Named Datatypes

Field	Description
_FILE	The filename to which the object belongs
_PATH	Full path to the group, dataset, or datatype within the file

Table 3-2: Additional Fields for Groups, Datasets, and Named Datatypes

Additional Fields for Groups

Field	Description
_COMMENT	Comment string

Table 3-3: Additional Fields for Groups

Additional Fields for Datasets, Attributes, and Named Datatypes

Field	Description
_DATATYPE	Datatype class, such as H5T_INTEGER
_STORAGESIZE	Size of each value in bytes
_PRECISION	Precision of each value in bits

Table 3-4: Additional Fields for Datasets, Attributes, and Named Datatypes

Field	Description
_SIGN	For integers, either 'signed' or 'unsigned'; otherwise a null string

Table 3-4: Additional Fields for Datasets, Attributes, and Named Datatypes

Additional Fields for Datasets and Attributes

Field	Description
_DATA	Data values stored in the object
_NDIMENSIONS	Number of dimensions in the dataspace
_DIMENSIONS	List of dataspace dimensions
_NELEMENTS	Total number of elements in the dataspace

Table 3-5: Additional Fields for Datasets and Attributes

Groups, datasets, datatypes, and attributes will be stored as substructures within *Result*. The tag names for these substructures are constructed from the actual object name by converting all non-alphanumeric characters to underscores, and converting all characters to uppercase.

Arguments

File

A string giving the name of the file to parse.

Loc_id

An integer giving the file or group identifier to access.

Name

A string giving the name of the group, dataset, or datatype within *Loc_id* to parse.

Keywords

FILE

Set this optional keyword to a string giving the filename associated with the Loc_id. This keyword is used for filling in the _FILE field within the returned structure, and is not required. The FILE keyword is ignored if the *File* argument is provided.

PATH

Set this optional keyword to a string giving the full path associated with the *Loc_id*. This keyword is used for filling in the _PATH field within the returned structure, and is not required. The PATH keyword is ignored if the *File* argument is provided.

READ_DATA

If this keyword is set, then all data from datasets is read in and stored in the returned structure. If READ_DATA is not provided then the _DATA field for datasets will be set to the string '<unread>'.

Note

For attribute objects all data is automatically read and stored in the structure.

Example

The following example shows how to parse a file, and then prints out the parsed structure.

```
File = FILEPATH('hdf5_test.h5', SUBDIR=['examples','data'])
Result = H5_PARSE(File)
help, Result, /STRUCTURE
```

When the above commands are entered, IDL prints:

```
** Structure <5f24468>, 13 tags, length=6872, data length=6664,
refs=1:
   _NAME STRING    'D:\RSI\idl56\examples\data\hdf5_test.h5'
   _ICONTYPE       STRING    'hdf'
   _TYPE STRING    'GROUP'
   _FILE STRING    'D:\RSI\idl56\examples\data\hdf5_test.h5'
   _PATH STRING    '/'
   _COMMENT        STRING    ''
   _2D_INT_ARRAY   STRUCT    -> <Anonymous> Array[1]
   A_NOTE          STRUCT    -> <Anonymous> Array[1]
   SL_TO_3D_INT_ARRAY
   STRUCT    -> <Anonymous> Array[1]
```

```
      ARRAYS          STRUCT    -> <Anonymous> Array[1]
      DATATYPES       STRUCT    -> <Anonymous> Array[1]
      IMAGES          STRUCT    -> <Anonymous> Array[1]
      LINKS           STRUCT    -> <Anonymous> Array[1]
```

Now print out the structure of a dataset within the "Images" group:

```
help, Result.images.eskimo, /STRUCTURE
```

IDL prints:

```
** Structure <16f1ca0>, 20 tags, length=840, data length=802,
refs=2:
   _NAME           STRING    'Eskimo'
   _ICONTYPE       STRING    'binary'
   _TYPE           STRING    'DATASET'
   _FILE           STRING
'D:\RSI\debug\examples\data\hdf5_test.h5'
   _PATH           STRING    '/images'
   _DATA           STRING    '<unread>'
   _NDIMENSIONS    LONG                 2
   _DIMENSIONS     ULONG64   Array[2]
   _NELEMENTS      ULONG64                   389400
   _DATATYPE       STRING    'H5T_INTEGER'
   _STORAGESIZE    ULONG                1
   _PRECISION      LONG                 8
   _SIGN           STRING    'unsigned'
   CLASS           STRUCT    -> <Anonymous> Array[1]
   IMAGE_VERSION   STRUCT    -> <Anonymous> Array[1]
   IMAGE_SUBCLASS  STRUCT    -> <Anonymous> Array[1]
   IMAGE_COLORMODEL
                   STRUCT    -> <Anonymous> Array[1]
   IMAGE_MINMAXRANGE
                   STRUCT    -> <Anonymous> Array[1]
   IMAGE_TRANSPARENCY
                   STRUCT    -> <Anonymous> Array[1]
   PALETTE         STRUCT    -> <Anonymous> Array[1]
```

Version History

Introduced 5.6

See Also

H5_BROWSER

H5A_CLOSE

The H5A_CLOSE procedure closes the specified attribute and releases resources used by it. After this routine is used, the attribute's identifier is no longer available until the H5A_OPEN routines are used again to specify that attribute. Further use of the attribute identifier is illegal.

Syntax

H5A_CLOSE, *Attribute_id*

Arguments

Attribute_id

An integer representing the attribute's identifier to be closed.

Keywords

None.

Version History

Introduced 5.6

See Also

H5A_OPEN_NAME, H5A_OPEN_IDX

H5A_GET_NAME

The H5A_GET_NAME function retrieves an attribute name given the attribute identifier number.

Syntax

Result = H5A_GET_NAME(*Attribute_id*)

Return Value

Returns a string containing the attribute name.

Arguments

Attribute_id

An integer representing the attribute's identifier to be queried.

Keywords

None.

Version History

Introduced 5.6

See Also

H5A_GET_SPACE, H5A_GET_TYPE

H5A_GET_NUM_ATTRS

The H5A_GET_NUM_ATTRS function returns the number of attributes attached to a group, dataset, or a named datatype.

Syntax

Result = H5A_GET_NUM_ATTRS(*Loc_id*)

Return Value

Returns an integer representing the number of attributes.

Arguments

Loc_id

An integer representing the identifier of the group, dataset, or named datatype to query.

Keywords

None.

Version History

Introduced 5.6

See Also

H5A_OPEN_IDX

H5A_GET_SPACE

The H5A_GET_SPACE function returns the identifier number of a copy of the dataspace for an attribute.

Syntax

Result = H5A_GET_SPACE(*Attribute_id*)

Return Value

Returns an integer representing the dataspace's identifier. This identifier can be released with the H5S_CLOSE.

Arguments

Attribute_id

An integer representing the attribute's identifier to be queried.

Keywords

None.

Version History

Introduced 5.6

See Also

H5A_GET_NAME, H5A_GET_TYPE, H5S_CLOSE

H5A_GET_TYPE

The H5A_GET_TYPE function returns the identifier number of a copy of the datatype for an attribute.

Syntax

Result = H5A_GET_TYPE(*Attribute_id*)

Return Value

Returns an integer representing the datatype identifier. This identifier should be released with the H5T_CLOSE.

Arguments

Attribute_id

An integer representing the attribute identifier to be queried.

Keywords

None.

Version History

Introduced 5.6

See Also

H5A_GET_SPACE, H5A_GET_NAME, H5T_CLOSE

H5A_OPEN_IDX

The H5A_OPEN_IDX function opens an existing attribute by the index of that attribute within an HDF5 file.

Syntax

Result = H5A_OPEN_IDX(*Loc_id*, *Index*)

Return Value

Returns an integer representing the attribute’s identifier number.

Arguments

Loc_id

An integer representing the identifier of the group, dataset, or named datatype containing the attribute within.

Index

An integer representing the zero-based index of the attribute to be accessed.

Keywords

None.

Version History

Introduced 5.6

See Also

H5A_OPEN_NAME, H5A_GET_NUM_ATTRS, H5A_GET_NAME, H5A_CLOSE

H5A_OPEN_NAME

The H5A_OPEN_NAME function opens an existing attribute by the name of that attribute within an HDF5 file.

Syntax

Result = H5A_OPEN_NAME(*Loc_id*, *Name*)

Return Value

Returns an integer representing the attribute's identifier number.

Arguments

Loc_id

An integer representing the identifier of the group, dataset, or named datatype containing the attribute within.

Name

A string representing the name of the attribute to be accessed.

Keywords

None.

Version History

Introduced 5.6

See Also

H5A_OPEN_IDX, H5A_CLOSE

H5A_READ

The H5A_READ function reads the data within an attribute, converting from the HDF5 file datatype into the HDF5 memory datatype, and finally into the corresponding IDL datatype.

Syntax

Result = H5A_READ(*Attribute_id*)

Return Value

Returns an IDL variable containing all of the attribute's data. For details on different return types and storage mechanisms, see the H5D_READ function.

Arguments

Attribute_id

An integer representing the attribute's identifier to be read.

Keywords

None.

Version History

Introduced 5.6

See Also

H5A_OPEN_NAME, H5A_OPEN_IDX, H5D_READ

H5D_CLOSE

The H5D_CLOSE procedure closes the specified dataset and releases its used resources. After this routine is used, the dataset's identifier is no longer available until the H5D_GET_SPACE is used again to specify that dataset.

Syntax

H5D_CLOSE, *Dataset_id*

Arguments

Dataset_id

An integer representing the dataset's identifier to be closed.

Keywords

None.

Version History

Introduced 5.6

See Also

H5D_OPEN

H5D_GET_SPACE

The H5D_GET_SPACE function returns an identifier number for a copy of the dataspace for a dataset.

Syntax

Result = H5D_GET_SPACE(*Dataset_id*)

Return Value

Returns an integer representing the dataspace's identifier. This identifier can be released with the H5S_CLOSE.

Arguments

Dataset_id

An integer representing the dataset's identifier to be queried.

Keywords

None.

Version History

Introduced 5.6

See Also

H5S_CLOSE, H5D_GET_STORAGE_SIZE, H5D_GET_TYPE

H5D_GET_STORAGE_SIZE

The H5D_GET_STORAGE_SIZE function returns the amount of storage in bytes required for a dataset. For chunked datasets, this value is the number of allocated chunks times the chunk size.

Note

This function does not typically need to be called, as IDL will automatically allocate the necessary memory when reading data.

Syntax

Result = H5D_GET_STORAGE_SIZE(*Dataset_id*)

Return Value

Returns an integer representing the amount of storage in bytes.

Arguments

Dataset_id

An integer representing the dataset's identifier to be queried.

Keywords

None.

Version History

Introduced 5.6

See Also

H5S_CLOSE, H5D_GET_SPACE, H5D_GET_TYPE

H5D_GET_TYPE

The H5D_GET_TYPE function returns an identifier number for a copy of the datatype for a dataset.

Syntax

Result = H5D_GET_TYPE(*Dataset_id*)

Return Value

Returns an integer representing the datatype's identifier. This identifier can be released with the H5T_CLOSE.

Arguments

Dataset_id

An integer representing the dataset's identifier to be queried.

Keywords

None.

Version History

Introduced 5.6

See Also

H5T_CLOSE, H5D_GET_SPACE, H5D_GET_STORAGE_SIZE

H5D_OPEN

The H5D_OPEN function opens an existing dataset within an HDF5 file.

Syntax

Result = H5D_OPEN(*Loc_id*, *Name*)

Return Value

Returns an integer representing the dataset's identifier. This identifier can be released with the H5D_CLOSE.

Arguments

Loc_id

An integer representing the identifier of the file or group containing the dataset.

Name

A string representing the name of the dataset to be accessed.

Keywords

None.

Version History

Introduced 5.6

See Also

H5D_CLOSE

H5D_READ

The H5D_READ function reads the data within a dataset, converting from the HDF5 file datatype into the HDF5 memory datatype, and finally into the corresponding IDL datatype.

Syntax

Result = H5D_READ(*Dataset_id* [, FILE_SPACE=*id*] [, MEMORY_SPACE=*id*])

Return Value

Returns an IDL variable containing the specified data. The different return types and storage mechanisms are described below.

Note

The dimensions for the *Result* variable are constructed using the following precedence rules:

If MEMORY_SPACE is specified, then the dimensions of the MEMORY_SPACE are used.

If only FILE_SPACE is specified, then the dimensions of the FILE_SPACE are used.

If neither MEMORY_SPACE nor FILE_SPACE are specified, then the dimensions are taken from the *Dataset_id*.

Arguments

Dataset_id

An integer representing the dataset's identifier to be read.

Keywords

FILE_SPACE

Set this keyword to the file dataspace identifier that should be used when reading the dataset. The FILE_SPACE keyword may be used to define hyperslabs or elements for subselection within the dataset. The default is zero (in HDF5 this is equivalent to H5S_ALL), which indicates that the entire dataspace should be read.

MEMORY_SPACE

Set this keyword to the memory dataspace identifier that should be used when copying the data from the file into memory. The MEMORY_SPACE keyword may be used to define hyperslabs or elements in which to place the data. The default is zero (in HDF5 this is equivalent to H5S_ALL), which indicates that the memory dataspace is identical to the file dataspace.

Return Type

When reading in HDF5 datasets, the datatype is first set to the native HDF5 types. These native types are then converted to IDL types as shown in the following table:

HDF5 Class	HDF5 Datatype	IDL Type
H5T_INTEGER H5T_BITFIELD H5T_ENUM	H5T_NATIVE_UINT8	Byte
	H5T_NATIVE_INT16	Integer
	H5T_NATIVE_UINT16	Unsigned integer
	H5T_NATIVE_INT32	Long integer
	H5T_NATIVE_UINT32	Unsigned long integer
	H5T_NATIVE_INT64	64-bit Integer
	H5T_NATIVE_UINT64	Unsigned 64-bit integer
H5T_REFERENCE	H5T_STD_REF_OBJ	Unsigned 64-bit integer
H5T_FLOAT	H5T_NATIVE_FLOAT	Floating point
	H5T_NATIVE_DOUBLE	Double-precision floating

Table 3-6: HDF and IDL Datatypes

HDF5 Class	HDF5 Datatype	IDL Type
H5T_STRING	H5T_C_S1	String
H5T_TIME	H5T_C_S1	String
H5T_COMPOUND	(Member datatypes)	Structure
H5T_ARRAY	(Super datatype)	(Super type)

Table 3-6: HDF and IDL Datatypes (Continued)

Note

Multidimensional datasets are returned in IDL row major order, with the fastest-varying dimensions listed first. HDF5 uses C column major order, with the fastest-varying dimensions listed last. In both cases, the memory layout for data elements is identical (i.e. no transpose is needed), only the order of the dimensions is reversed.

Note

For the H5T_ARRAY datatype the array dimensions are concatenated with the dataset dimensions, with the array dimensions varying more rapidly.

Note

Structure tag names are constructed from H5T_COMPOUND member names by switching to uppercase and converting all non-alphanumeric characters to underscores.

Version History

Introduced 5.6

See Also

H5D_CLOSE, H5D_OPEN, H5A_READ, H5S_CREATE_SIMPLE, H5S_SELECT_ELEMENTS, H5S_SELECT_HYPERSLAB

H5F_CLOSE

The H5F_CLOSE procedure closes the specified file and releases resources used by it. After this routine is used, the file's identifier is no longer available until the H5F_CLOSE routine is used again to specify that file.

Syntax

H5F_CLOSE, *File_id*

Arguments

File_id

An integer representing the file's identifier to be closed.

Keywords

None.

Version History

Introduced 5.6

See Also

H5F_OPEN

H5F_IS_HDF5

The H5F_IS_HDF5 function determines if a file is in the HDF5 format.

Syntax

Result = H5F_IS_HDF5(*Filename*)

Return Value

Returns an integer of 1 if the file is in the HDF5 format, 0 if otherwise.

Arguments

Filename

A string representing the name of the files to be checked.

Keywords

None.

Version History

Introduced 5.6

See Also

H5F_OPEN

H5F_OPEN

The H5F_OPEN function opens an existing HDF5 file.

Syntax

Result = H5F_OPEN(*Filename*)

Return Value

Returns an integer representing the file identifier number. This identifier can be released with the H5F_CLOSE.

Arguments

Filename

A string representing the name of the file to be accessed.

Keywords

None.

Version History

Introduced 5.6

See Also

H5F_CLOSE, H5F_IS_HDF5

H5G_CLOSE

The H5G_CLOSE procedure closes the specified group and releases resources used by it. After this routine is used, the group's identifier is no longer available until the H5F_OPEN routine is used again to specify that group.

Syntax

H5G_CLOSE, *Group_id*

Arguments

Group_id

An integer representing the group's identifier to be closed.

Keywords

None.

Version History

Introduced 5.6

See Also

H5G_OPEN

H5G_GET_COMMENT

The H5G_GET_COMMENT function retrieves a comment string from a specified object.

Syntax

Result = H5G_GET_COMMENT(*Loc_id*, *Name*)

Return Value

Returns a string containing the comment, or a null string if no comment exists.

Arguments

Loc_id

An integer representing the identifier of the file or group.

Name

A string representing the name of the object for which to retrieve the comment.

Keywords

None.

Version History

Introduced 5.6

See Also

H5G_GET_OBJINFO

H5G_GET_LINKVAL

The H5G_GET_LINKVAL function returns the name of the object pointed to by a symbolic link.

Syntax

Result = H5G_GET_LINKVAL(*Loc_id*, *Name*)

Return Value

Returns a string containing the name of the object pointed to by a symbolic link.

Arguments

Loc_id

An integer representing the identifier of the file or group.

Name

A string representing the name of the symbolic link for which to retrieve the link value.

Keywords

None.

Version History

Introduced 5.6

See Also

H5G_GET_OBJINFO

H5G_GET_MEMBER_NAME

The H5G_GET_MEMBER_NAME function retrieves the name of an object within a group, by its zero-based index.

Note

This function is not part of the standard HDF5 interface, but is provided as a programming convenience. The H5Giterate() C function is used to retrieve the name.

Syntax

Result = H5G_GET_MEMBER_NAME(*Loc_id*, *Name*, *Index*)

Return Value

Returns a string containing the object's name.

Arguments

Loc_id

An integer representing the identifier of the file or group.

Name

A string representing the name of the group in which to retrieve the member name.

Index

An integer representing the zero-based index of the object for which to retrieve the name.

Keywords

None.

Version History

Introduced 5.6

See Also

H5G_GET_NMEMBERS

H5G_GET_NMEMBERS

The H5G_GET_NMEMBERS function returns the number of objects within a group.

Note

This function is not part of the standard HDF5 interface, but is provided as a programming convenience. The H5Giterate() C function is used to retrieve the number of members.

Syntax

Result = H5G_GET_NMEMBERS(*Loc_id*, *Name*)

Return Value

Returns an integer representing the number of objects.

Arguments

Loc_id

An integer representing the identifier of the file or group.

Name

A string representing the name of the group for which to retrieve the number of members.

Keywords

None.

Version History

Introduced 5.6

See Also

H5G_GET_MEMBER_NAME

H5G_GET_OBJINFO

The H5G_GET_OBJINFO function retrieves information from a specified object.

Syntax

Result = H5G_GET_OBJINFO(*Loc_id*, *Name* [, /FOLLOW_LINK])

Return Value

Returns a structure of the name H5F_STAT containing the following fields:

FILENO

This field contains two integers which, along with the OBJNO field, uniquely identify the object among all open HDF5 files.

OBJNO

This field contains two integers which, along with the FILENO field, uniquely identify the object among all open HDF5 files. If all four values in FILENO and OBJNO are the same between two objects, then these two objects are the same.

NLINK

The number of hard links to the object. If this field is zero, then the object is a symbolic link.

TYPE

A string representing the object type. Possible values are:

- ‘LINK’
- ‘GROUP’
- ‘DATASET’
- ‘TYPE’
- ‘UNKNOWN’

MTIME

The modification time for the object, in seconds since 1 January 1970.

Tip

You can convert the MTIME field from seconds to a date/time string using SYSTIME(*0*, *mtime*). See SYSTIME for more information.

LINKLEN

If the object is a symbolic link (and the FOLLOW_LINK keyword is not set), then this field will contain the length in characters of the link value. The link value itself may be retrieved using H5D_GET_LINKVAL.

Arguments

Loc_id

An integer representing the identifier of the file or group.

Name

A string representing the name of the object for which to retrieve the information structure.

Keywords

FOLLOW_LINK

If *Name* is a symbolic link, then set this keyword to follow the symbolic link and retrieve information about the linked object. The default is to return information about the symbolic link itself.

Version History

Introduced 5.6

See Also

H5G_GET_LINKVAL

H5G_OPEN

The H5G_OPEN function opens an existing group within an HDF5 file.

Syntax

Result = H5G_OPEN(*Loc_id*, *Name*)

Return Value

Returns an integer representing the group's identifier number. This identifier can be released with the H5G_CLOSE.

Arguments

Loc_id

An integer representing the identifier of the file or group containing the group to be accessed.

Name

A string representing the name of the group to be accessed.

Keywords

None.

Version History

Introduced 5.6

See Also

H5G_CLOSE

H5I_GET_TYPE

The H5I_GET_TYPE function returns the object's type.

Syntax

Result = H5I_GET_TYPE(*Obj_id*)

Return Value

Returns a string representing the object type. Possible return values include:

- 'FILE'
- 'GROUP'
- 'DATATYPE'
- 'DATASPACE'
- 'DATASET'
- 'ATTR'
- 'BADID'

Arguments

Obj_id

An integer representing the object's identifier for which to return the type.

Keywords

None.

Version History

Introduced 5.6

H5R_DEREFERENCE

The H5R_DEREFERENCE function opens a reference and returns the object identifier.

Syntax

Result = H5R_DEREFERENCE(*Loc_id*, *Reference*)

Return Value

The *Result* is an integer giving the identifier number. This identifier should be released using the appropriate close procedure.

Arguments

Loc_id

An integer giving the identifier in which the reference dataset is located.

Reference

An integer giving the reference number to open.

Keywords

None.

Version History

Introduced 5.6

See Also

H5R_GET_OBJECT_TYPE

H5R_GET_OBJECT_TYPE

The H5R_GET_OBJECT_TYPE function returns the type of object that an object reference points to.

Syntax

Result = H5R_GET_OBJECT_TYPE(*Loc_id*, *Reference*)

Return Value

The *Result* is a string giving the object type. Possible return values include:

- ‘FILE’
- ‘GROUP’
- ‘DATASET’
- ‘DATASPACE’
- ‘DATASET’
- ‘ATTR’
- ‘BADID’

Arguments

Loc_id

An integer giving the identifier in which the reference dataset is located.

Reference

An integer giving the reference number to query.

Keywords

None.

Version History

Introduced 5.6

See Also

H5R_DEREFERENCE

H5S_CLOSE

The H5S_CLOSE procedure releases and terminates access to a dataspace. After this routine is used, the dataspace's identifier is no longer available.

Warning

Failure to release a dataspace using this procedure will result in resource leaks.

Syntax

H5S_CLOSE, *Dataspace_id*

Arguments

Dataspace_id

An integer representing the dataspace's identifier to close.

Keywords

None.

Version History

Introduced 5.6

See Also

H5D_GET_SPACE

H5S_COPY

The H5S_COPY function copies an existing dataspace.

Syntax

Result = H5S_COPY(*Dataspace_id*)

Return Value

Returns an integer representing the dataspace's identifier number. The dataspace identifier can be released with the H5S_CLOSE.

Arguments

Dataspace_id

An integer representing the dataspace identifier to copy.

Keywords

None.

Version History

Introduced 5.6

See Also

H5S_CREATE_SIMPLE, H5S_CLOSE

H5S_CREATE_SIMPLE

The H5S_CREATE_SIMPLE function creates a simple dataspace.

Syntax

Result = H5S_CREATE_SIMPLE(*Dimensions* [, MAX_DIMENSIONS=*vector*])

Return Value

Returns an integer representing the dataspace's identifier number. This dataspace identifier can be released with the H5S_CLOSE.

Arguments

Dimensions

Set this argument to a vector containing the dimensions for the dataspace.

Note

The *Dimensions* argument should be specified in IDL's row-major order. Internally, the dimensions will be reversed to match HDF5/C's column-major order.

Keywords

MAX_DIMENSIONS

Set this keyword to a vector containing the maximum dimensions for the dataspace. The MAX_DIMENSIONS must have the same number of elements as the *Dimensions* argument. If MAX_DIMENSIONS is omitted then the maximum dimensions are set to *Dimensions*. You can use a value of -1 in MAX_DIMENSIONS to indicate an unlimited dimension.

Note

The MAX_DIMENSIONS keyword should be specified in IDL's row-major order. Internally, the dimensions will be reversed to match HDF5/C's column-major order.

Version History

Introduced 5.6

See Also

H5S_CLOSE, H5S_COPY

H5S_GET_SELECT_BOUNDS

The H5S_GET_SELECT_BOUNDS function retrieves the coordinates of the bounding box containing the current dataspace selection.

Syntax

Result = H5S_GET_SELECT_BOUNDS(*Dataspace_id*)

Return Value

Returns an (*m* x 2) array, where *m* is the number of dimensions (or rank) of the dataspace. The first row in the array is the starting coordinates of the bounding box, while the second row is the ending coordinates.

Arguments

Dataspace_id

An integer representing the dataspace's identifier to be queried.

Keywords

None.

Version History

Introduced 5.6

See Also

H5S_GET_SIMPLE_EXTENT_NPOINTS, H5S_GET_SELECT_NPOINTS, H5S_GET_SELECT_ELEM_NPOINTS, H5S_GET_SELECT_HYPER_NBLOCKS

H5S_GET_SELECT_ELEM_NPOINTS

The H5S_GET_SELECT_ELEM_NPOINTS function determines the number of element points in the current dataspace selection.

Syntax

Result = H5S_GET_SELECT_ELEM_NPOINTS(*Dataspace_id*)

Return Value

Returns an integer representing the number of element points.

Arguments

Dataspace_id

An integer representing the dataspace's identifier to be queried.

Keywords

None.

Version History

Introduced 5.6

See Also

H5S_GET_SELECT_BOUNDS, H5S_GET_SELECT_HYPER_NBLOCKS, H5S_GET_SELECT_NPOINTS, H5S_GET_SIMPLE_EXTENT_NPOINTS

H5S_GET_SELECT_ELEM_POINTLIST

The H5S_GET_SELECT_ELEM_POINTLIST function returns a list of the element points in the current dataspace selection.

Syntax

Result = H5S_GET_SELECT_ELEM_POINTLIST(*Dataspace_id* [, START=*value*] [, NUMBER=*value*])

Return Value

The *Result* is an (*m* x *n*) array, where *m* is the number of dimensions (or rank) of the dataspace, and *n* is the number of selected points. Each row contains the coordinates for an element selection point.

Arguments

Dataspace_id

An integer representing the dataspace's identifier to be queried.

Keywords

START

Set this keyword to an integer representing the point to start with, counting from 0. The default is START = 0.

NUMBER

Set this keyword to an integer representing the number of element points to return. The default is NUMBER = (N - START), where N is the total number of element points in the selection.

Version History

Introduced 5.6

See Also

H5S_GET_SELECT_ELEM_NPOINTS, H5S_GET_SELECT_NPOINTS

H5S_GET_SELECT_HYPER_BLOCKLIST

The H5S_GET_SELECT_HYPER_BLOCKLIST function returns a list of the hyperslab blocks in the current dataspace selection.

Syntax

Result = H5S_GET_SELECT_HYPER_BLOCKLIST(*Dataspace_id* [, START=*value*] [, NUMBER=*value*])

Return Value

Returns an (*m* x 2*n*) array, where *m* is the number of dimensions (or rank) of the dataspace. The 2*n* rows of *Result* contain the list of blocks. The first row contains the start coordinates of the first block, followed by the next row which contains the opposite corner coordinates, followed by the next row which contains the start coordinates of the second block, etc.

Arguments

Dataspace_id

An integer representing the dataspace's identifier to be queried.

Keywords

START

Set this keyword to an integer representing the block to start with, counting from 0. The default is START = 0.

NUMBER

Set this keyword to an integer representing the number of blocks to return. The default is NUMBER = (N - START), where N is the total number of blocks in the selection.

Version History

Introduced 5.6

See Also

H5S_GET_SELECT_HYPER_NBLOCKS, H5S_GET_SELECT_NPOINTS

H5S_GET_SELECT_HYPER_NBLOCKS

The H5S_GET_SELECT_HYPER_NBLOCKS function determines the number of hyperslab blocks in the current dataspace selection.

Syntax

Result = H5S_GET_SELECT_HYPER_NBLOCKS(*Dataspace_id*)

Return Value

Returns an integer representing the number of blocks.

Arguments

Dataspace_id

An integer representing the dataspace identifier to be queried.

Keywords

None.

Version History

Introduced 5.6

See Also

H5S_GET_SELECT_BOUNDS, H5S_GET_SELECT_ELEM_NPOINTS, H5S_GET_SELECT_NPOINTS, H5S_GET_SIMPLE_EXTENT_NPOINTS

H5S_GET_SELECT_NPOINTS

The H5S_GET_SELECT_NPOINTS function determines the number of elements in a dataspace selection.

Syntax

Result = H5S_GET_SELECT_NPOINTS(*Dataspace_id*)

Return Value

Returns an integer representing the number of elements.

Arguments

Dataspace_id

An integer representing the dataspace identifier to be queried.

Keywords

None.

Version History

Introduced 5.6

See Also

H5S_GET_SELECT_BOUNDS, H5S_GET_SELECT_ELEM_NPOINTS,
H5S_GET_SELECT_HYPER_NBLOCKS,
H5S_GET_SIMPLE_EXTENT_NPOINTS

H5S_GET_SIMPLE_EXTENT_DIMS

The H5S_GET_SIMPLE_EXTENT_DIMS function returns the dimension sizes for a dataspace.

Syntax

Result = H5S_GET_SIMPLE_EXTENT_DIMS(*Dataspace_id*
[, MAX_DIMENSIONS=*variable*])

Return Value

Returns a vector containing the dimension sizes.

Arguments

Dataspace_id

An integer representing the dataspace’s identifier to be queried.

Keywords

MAX_DIMENSIONS

Set this keyword to a named variable to return the maximum dimension sizes for the dataspace.

Version History

Introduced 5.6

See Also

H5S_GET_SIMPLE_EXTENT_NDIMS,
H5S_GET_SIMPLE_EXTENT_NPOINTS, H5S_GET_SIMPLE_EXTENT_TYPE

H5S_GET_SIMPLE_EXTENT_NDIMS

The H5S_GET_SIMPLE_EXTENT_NDIMS function determines the number of dimensions (or rank) of a dataspace.

Syntax

Result = H5S_GET_SIMPLE_EXTENT_NDIMS(*Dataspace_id*)

Return Value

Returns an integer representing the number of dimensions.

Arguments

Dataspace_id

An integer representing the dataspace's identifier to be queried.

Keywords

None.

Version History

Introduced 5.6

See Also

H5S_GET_SIMPLE_EXTENT_DIMS, H5S_GET_SIMPLE_EXTENT_NPOINTS, H5S_GET_SIMPLE_EXTENT_TYPE

H5S_GET_SIMPLE_EXTENT_NPOINTS

The H5S_GET_SIMPLE_EXTENT_NPOINTS function determines the number of elements in a dataspace.

Syntax

Result = H5S_GET_SIMPLE_EXTENT_NPOINTS(*Dataspace_id*)

Return Value

Returns an integer representing the number of elements.

Arguments

Dataspace_id

An integer representing the dataspace's identifier to be queried.

Keywords

None.

Version History

Introduced 5.6

See Also

H5S_GET_SIMPLE_EXTENT_DIMS, H5S_GET_SIMPLE_EXTENT_NDIMS, H5S_GET_SIMPLE_EXTENT_TYPE

H5S_GET_SIMPLE_EXTENT_TYPE

The H5S_GET_SIMPLE_EXTENT_TYPE function returns the current class of a dataspace.

Syntax

Result = H5S_GET_SIMPLE_EXTENT_TYPE(*Dataspace_id*)

Return Value

Returns a string containing the class. Possible values are:

- 'H5S_SCALAR'
- 'H5S_SIMPLE'
- 'H5S_COMPLEX'
- 'H5S_NO_CLASS'

Arguments

Dataspace_id

An integer representing the dataspace's identifier to be queried.

Keywords

None.

Version History

Introduced 5.6

See Also

H5S_GET_SIMPLE_EXTENT_DIMS, H5S_GET_SIMPLE_EXTENT_NDIMS, H5S_GET_SIMPLE_EXTENT_NPOINTS

H5S_IS_SIMPLE

The H5S_IS_SIMPLE function determines whether a dataspace is a simple dataspace.

Syntax

Result = H5S_IS_SIMPLE(*Dataspace_id*)

Return Value

Returns an integer of 1 if the dataspace is simple and 0 if it is not.

Arguments

Dataspace_id

An integer representing the dataspace's identifier to be queried.

Keywords

None.

Version History

Introduced 5.6

H5S_OFFSET_SIMPLE

The H5S_OFFSET_SIMPLE procedure sets the selection offset for a simple dataspace. The offset allows the same shaped selection to be moved to different locations within the dataspace.

Syntax

H5S_OFFSET_SIMPLE, *Dataspace_id*, *Offset*

Arguments

Dataspace_id

An integer representing the dataspace's identifier on which to set the selection offset.

Offset

An *m*-element vector of integers, where *m* is the number of dataspace dimensions, containing the offsets.

Keywords

None.

Version History

Introduced 5.6

See Also

H5S_GET_SELECT_BOUNDS, H5S_SELECT_ELEMENTS, H5S_SELECT_HYPERSLAB

H5S_SELECT_ALL

The H5S_SELECT_ALL procedure selects the entire extent of a dataspace.

Syntax

H5S_SELECT_ALL, *Dataspace_id*

Arguments

Dataspace_id

An integer representing the dataspace's identifier to be selected.

Keywords

None.

Version History

Introduced 5.6

See Also

H5S_GET_SELECT_NPOINTS, H5S_SELECT_ELEMENTS, H5S_SELECT_HYPERSLAB, H5S_SELECT_NONE

H5S_SELECT_ELEMENTS

The H5S_SELECT_ELEMENTS procedure selects array elements to be included in the selection for a dataspace.

Syntax

H5S_SELECT_ELEMENTS, *Dataspace_id*, *Coordinates*, /RESET

Arguments

Dataspace_id

An integer representing the dataspace's identifier on which to set the selection.

Coordinates

An *m*-element vector, or an (*m* x *n*) array, where *m* is the number of dimensions (or rank) of the dataspace, and *n* is the number of selected points. Each row contains the coordinates for an element selection point.

Keywords

RESET

Set this keyword to replace the existing selection with the new *Coordinates*. The default is RESET = 0 which adds the new selection to the existing selection.

Note

The RESET keyword must be set (/RESET or RESET = 1) or the H5S_SELECT_ELEMENTS routine will result in an error message. This error message comes from the HDF5 library, which forces a default of RESET = 0 but insists on this keyword being set for this routine to work.

Version History

Introduced 5.6

See Also

H5S_GET_SELECT_ELEM_NPOINTS, H5S_GET_SELECT_ELEM_POINTLIST, H5S_GET_SELECT_NPOINTS, H5S_SELECT_HYPERSLAB

H5S_SELECT_HYPERSLAB

The H5S_SELECT_HYPERSLAB procedure selects a hyperslab region to be included in the selection for a dataspace.

Note

If all of the elements in the selected hyperslab region are already selected, then a new hyperslab region is not created.

Syntax

H5S_SELECT_HYPERSLAB, *Dataspace_id*, *Start*, *Count*, [, BLOCK=*vector*] [, /RESET] [, STRIDE=*vector*]

Arguments

Dataspace_id

An integer representing the dataspace's identifier on which to set the selection.

Start

An *m*-element vector of integers, where *m* is the number of dataspace dimensions, containing the starting location for the hyperslab.

Count

An *m*-element vector of integers containing the number of blocks to select in each dimension.

Keywords

BLOCK

Set this keyword to an *m*-element vector of integers containing the size of a block. The default is a single element in each dimension (for example BLOCK is set to a vector of all 1's).

RESET

Set this keyword to replace the existing selection with the new selection. The default is RESET=0 which adds the new selection to the existing selection.

STRIDE

Set this keyword to an *m*-element vector of integers containing the number of elements to move in each dimension when selecting blocks. The default is to move a single element in each dimension (for example STRIDE is set to a vector of all 1's). STRIDE values must be greater than zero.

Version History

Introduced 5.6

See Also

H5S_GET_SELECT_HYPER_BLOCKLIST, H5S_GET_SELECT_HYPER_NBLOCKS, H5S_GET_SELECT_NPOINTS, H5S_SELECT_ELEMENTS

H5S_SELECT_NONE

The H5S_SELECT_NONE procedure resets the dataspace selection region to include no elements.

Syntax

H5S_SELECT_NONE, *Dataspace_id*

Arguments

Dataspace_id

An integer representing the dataspace's identifier to be reset.

Keywords

None.

Version History

Introduced 5.6

See Also

H5S_GET_SELECT_NPOINTS, H5S_SELECT_ALL, H5S_SELECT_ELEMENTS, H5S_SELECT_HYPERSLAB

H5S_SELECT_VALID

The H5S_SELECT_VALID function verifies that the selection is within the extent of a dataspace.

Syntax

Result = H5S_SELECT_VALID(*Dataspace_id*)

Return Value

Returns an integer of 1 if the selection is within the dataspace and 0 if it is not.

Arguments

Dataspace_id

An integer representing the dataspace's identifier to be queried.

Keywords

None.

Version History

Introduced 5.6

See Also

H5S_GET_SELECT_NPOINTS, H5S_SELECT_ELEMENTS, H5S_SELECT_HYPERSLAB

H5T_CLOSE

The H5T_CLOSE procedure releases the specified datatype's identifier and releases resources used by it. After this routine is used, the datatype's identifier is no longer available until the H5T_OPEN routine is used again to specify that datatype.

Syntax

H5T_CLOSE, *Datatype_id*

Arguments

Datatype_id

An integer representing the datatype's identifier to be closed.

Keywords

None.

Version History

Introduced 5.6

See Also

H5T_OPEN

H5T_COMMITTED

The H5T_COMMITTED function determines whether a datatype is a named datatype or a transient type.

Syntax

Result = H5T_COMMITTED(*Datatype_id*)

Return Value

Returns an integer of 1 if the datatype is named and 0 if the datatype is transient.

Arguments

Datatype_id

An integer representing the datatyped identifier to be queried.

Keywords

None.

Version History

Introduced 5.6

H5T_COPY

The H5T_COPY function copies an existing datatype. The returned type is transient and unlocked.

Syntax

Result = H5T_COPY(*Datatype_id*)

Return Value

Returns an integer representing the datatype's identifier number. This identifier can be released with the H5T_CLOSE procedure.

Arguments

Datatype_id

An integer representing the datatype's identifier to be copied.

Keywords

None.

Version History

Introduced 5.6

See Also

H5T_CLOSE, H5T_OPEN

H5T_EQUAL

The H5T_EQUAL function determines whether two datatype identifiers refer to the same datatype.

Syntax

Result = H5T_EQUAL(*Datatype_id1*, *Datatype_id2*)

Return Value

Returns an integer of 1 if the identifiers refer to the same datatype and 0 if they do not.

Arguments

Datatype_id1

An integer representing the first datatype identifier.

Datatype_id2

An integer representing the second datatype identifier.

Keywords

None.

Version History

Introduced 5.6

See Also

H5T_COPY

H5T_GET_ARRAY_DIMS

The H5T_GET_ARRAY_DIMS function returns the dimension sizes for an array datatype object.

Syntax

Result = H5T_GET_ARRAY_DIMS(*Datatype_id* [, PERMUTATIONS=*variable*])

Return Value

Returns a vector containing the dimension sizes.

Arguments

Datatype_id

An integer representing the datatype's identifier to be queried.

Keywords

PERMUTATIONS

Set this keyword to a named variable in which to return the dimension permutations (C versus FORTRAN).

Version History

Introduced 5.6

See Also

H5T_GET_ARRAY_NDIMS

H5T_GET_ARRAY_NDIMS

The H5T_GET_ARRAY_NDIMS function determines the number of dimensions (or rank) of an array datatype object.

Syntax

Result = H5T_GET_ARRAY_NDIMS(*Datatype_id*)

Return Value

Returns an integer representing the number of dimensions.

Arguments

Datatype_id

An integer representing the datatype's identifier to be queried.

Keywords

None.

Version History

Introduced 5.6

See Also

H5T_GET_ARRAY_DIMS

H5T_GET_CLASS

The H5T_GET_CLASS function returns the datatype's class.

Syntax

Result = H5T_GET_CLASS(*Datatype_id*)

Return Value

Returns a string containing the datatype's class. Possible return values include:

- 'H5T_INTEGER'
- 'H5T_FLOAT'
- 'H5T_TIME'
- 'H5T_STRING'
- 'H5T_BITFIELD'
- 'H5T_OPAQUE'
- 'H5T_COMPOUND'
- 'H5T_REFERENCE'
- 'H5T_ENUM'
- 'H5T_VLEN'
- 'H5T_ARRAY'
- 'H5T_NO_CLASS'

Arguments

Datatype_id

An integer representing the datatype's identifier to be queried.

Keywords

None.

Version History

Introduced 5.6

See Also

H5T_GET_SIZE, H5T_GET_SUPER

H5T_GET_CSET

The H5T_GET_CSET function returns the character set type of a string datatype.

Syntax

Result = H5T_GET_CSET(*Datatype_id*)

Return Value

Returns a string containing the character set type. Possible values are:

- ‘ASCII’ — US ASCII
- ‘ERROR’

Arguments

Datatype_id

An integer representing the datatype’s identifier to be queried.

Keywords

None.

Version History

Introduced 5.6

H5T_GET_EBIAS

The H5T_GET_EBIAS function returns the exponent bias of a floating-point type.

Syntax

Result = H5T_GET_EBIAS(*Datatype_id*)

Return Value

Returns an integer representing the exponent bias.

Arguments

Datatype_id

An integer representing the datatype's identifier to be queried.

Keywords

None.

Version History

Introduced 5.6

See Also

H5T_GET_FIELDS

H5T_GET_FIELDS

The H5T_GET_FIELDS function retrieves information about the positions and sizes of bit fields within a floating-point datatype.

Syntax

Result = H5T_GET_FIELDS(*Datatype_id*)

Return Value

Returns a structure named H5T_GET_FIELDS containing the following tags:

TYPE_ID

The datatype's identifier *Datatype_id*.

SIGN_POS

The position of the floating-point sign bit.

EXP_POS

The bit position of the exponent.

EXP_SIZE

The size of the exponent in bits.

MAN_POS

The bit position of the mantissa.

MAN_SIZE

The size of the mantissa in bits.

Arguments

Datatype_id

An integer representing the datatype's identifier to be queried.

Keywords

None.

Version History

Introduced 5.6

See Also

H5T_GET_EBIAS, H5T_GET_INPAD, H5T_GET_NORM, H5T_GET_OFFSET, H5T_GET_ORDER, H5T_GET_PAD, H5T_GET_PRECISION

H5T_GET_INPAD

The H5T_GET_INPAD function returns the padding method for unused internal bits within a floating-point datatype.

Syntax

Result = H5T_GET_INPAD(*Datatype_id*)

Return Value

Returns an integer representing the padding method. Possible values are:

- 0 — Background set to zeroes
- 1 — Background set to ones
- 2 — Background left unchanged

Arguments

Datatype_id

An integer representing the datatype identifier to be queried.

Keywords

None.

Version History

Introduced 5.6

See Also

H5T_GET_FIELDS

H5T_GET_MEMBER_CLASS

The H5T_GET_MEMBER_CLASS function returns the datatype class of a compound datatype member.

Syntax

Result = H5T_GET_MEMBER_CLASS(*Datatype_id*, *Member*)

Return Value

Returns a string containing the datatype class. Possible values are:

- ‘H5T_INTEGER’
- ‘H5T_FLOAT’
- ‘H5T_TIME’
- ‘H5T_STRING’
- ‘H5T_BITFIELD’
- ‘H5T_OPAQUE’
- ‘H5T_COMPOUND’
- ‘H5T_REFERENCE’
- ‘H5T_ENUM’
- ‘H5T_VLEN’
- ‘H5T_ARRAY’
- ‘H5T_NO_CLASS’

Arguments

Datatype_id

An integer representing the datatype’s identifier to be queried.

Member

An integer representing the member index, starting at zero.

Keywords

None.

Version History

Introduced 5.6

See Also

H5T_GET_MEMBER_NAME, H5T_GET_MEMBER_OFFSET,
H5T_GET_MEMBER_TYPE, H5T_GET_NMEMBERS

H5T_GET_MEMBER_NAME

The H5T_GET_MEMBER_NAME function returns the datatype name of a compound datatype member.

Syntax

Result = H5T_GET_MEMBER_NAME(*Datatype_id*, *Member*)

Return Value

Returns a string containing the datatype name.

Arguments

Datatype_id

An integer representing the datatype's identifier to be queried.

Member

An integer representing the member index, starting at zero.

Keywords

None.

Version History

Introduced 5.6

See Also

H5T_GET_MEMBER_CLASS, H5T_GET_MEMBER_OFFSET, H5T_GET_MEMBER_TYPE, H5T_GET_NMEMBERS

H5T_GET_MEMBER_OFFSET

The H5T_GET_MEMBER_OFFSET function returns the byte offset of a field within a compound datatype.

Syntax

Result = H5T_GET_MEMBER_OFFSET(*Datatype_id*, *Member*)

Return Value

Returns an integer representing the byte offset.

Arguments

Datatype_id

An integer representing the datatype's identifier to be queried.

Member

An integer representing the member index, starting at zero.

Keywords

None.

Version History

Introduced 5.6

See Also

H5T_GET_MEMBER_CLASS, H5T_GET_MEMBER_NAME, H5T_GET_MEMBER_TYPE, H5T_GET_NMEMBERS

H5T_GET_MEMBER_TYPE

The H5T_GET_MEMBER_TYPE function returns the datatype identifier for a specified member within a compound datatype.

Syntax

Result = H5T_GET_MEMBER_TYPE(*Datatype_id*, *Member*)

Return Value

Returns an integer representing the datatype identifier. This identifier should be closed using H5T_CLOSE.

Arguments

Datatype_id

An integer representing the datatype's identifier to be queried.

Member

An integer representing the member index, starting at zero.

Keywords

None.

Version History

Introduced 5.6

See Also

H5T_GET_MEMBER_CLASS, H5T_GET_MEMBER_NAME, H5T_GET_MEMBER_OFFSET, H5T_CLOSE, H5T_GET_NMEMBERS

H5T_GET_NMEMBERS

The H5T_GET_NMEMBERS function returns the number of fields in a compound datatype.

Syntax

Result = H5T_GET_NMEMBERS(*Datatype_id*)

Return Value

Returns an integer representing the number of fields.

Arguments

Datatype_id

An integer representing the datatype's identifier to be queried.

Keywords

None.

Version History

Introduced 5.6

See Also

H5T_GET_MEMBER_CLASS, H5T_GET_MEMBER_NAME, H5T_GET_MEMBER_OFFSET, H5T_GET_MEMBER_TYPE

H5T_GET_NORM

The H5T_GET_NORM function returns the mantissa normalization of a floating-point datatype.

Syntax

Result = H5T_GET_NORM(*Datatype_id*)

Return Value

Returns a string containing the mantissa normalization. Possible values are:

- ‘IMPLIED’ — Most-significant bit of mantissa not stored, always 1
- ‘MSBSET’ — Most-significant bit of mantissa is always 1
- ‘NORM’ — Mantissa is not normalized
- ‘ERROR’

Arguments

Datatype_id

An integer representing the datatype’s identifier to be queried.

Keywords

None.

Version History

Introduced 5.6

See Also

H5T_GET_FIELDS

H5T_GET_OFFSET

The H5T_GET_OFFSET function returns the bit offset of the first significant bit in an atomic datatype. The offset is the number of bits of padding that follows the significant bits (for big endian) or precedes the significant bits (for little endian).

Syntax

Result = H5T_GET_OFFSET(*Datatype_id*)

Return Value

Returns an integer representing the bit offset.

Arguments

Datatype_id

An integer representing the datatype's identifier to be queried.

Keywords

None.

Version History

Introduced 5.6

See Also

H5T_GET_FIELDS

H5T_GET_ORDER

The H5T_GET_ORDER function returns the byte order of an atomic datatype.

Syntax

Result = H5T_GET_ORDER(*Datatype_id*)

Return Value

Returns a string representing the byte order. Possible values are:

- ‘LE’ — Little endian
- ‘BE’ — Big endian
- ‘VAX’ — VAX mixed ordering
- ‘NONE’
- ‘ERROR’

Arguments

Datatype_id

An integer representing the datatype’s identifier to be queried.

Keywords

None.

Version History

Introduced 5.6

See Also

H5T_GET_INPAD, H5T_GET_PAD, H5T_GET_PRECISION

H5T_GET_PAD

The H5T_GET_PAD function returns the padding method of the least significant bit (*lsb*) and most significant bit (*msb*) of an atomic datatype.

Syntax

Result = H5T_GET_PAD(*Datatype_id*)

Return Value

Returns a two-element vector [*lsb*, *msb*]. Possible values are:

- 0 — Background set to zeroes
- 1 — Background set to ones
- 2 — Background left unchanged.

Arguments

Datatype_id

An integer representing the datatype’s identifier to be queried.

Keywords

None.

Version History

Introduced 5.6

See Also

H5T_GET_INPAD, H5T_GET_ORDER, H5T_GET_PRECISION

H5T_GET_PRECISION

The H5T_GET_PRECISION function returns the precision in bits of an atomic datatype. The precision is the number of significant bits which, unless padded, is 8 times larger than the byte size from H5T_GET_CSET.

Syntax

Result = H5T_GET_PRECISION(*Datatype_id*)

Return Value

Returns an integer representing the bit precision.

Arguments

Datatype_id

An integer representing the datatype’s identifier to be queried.

Keywords

None.

Version History

Introduced 5.6

See Also

H5T_GET_INPAD, H5T_GET_ORDER, H5T_GET_PAD, H5T_GET_SIZE

H5T_GET_SIGN

The H5T_GET_SIGN function returns the sign type for an integer datatype.

Syntax

Result = H5T_GET_SIGN(*Datatype_id*)

Return Value

Returns an integer representing the sign type. Possible values are:

- -1 — Error
- 0 — Unsigned integer type
- 1 — Two's complement signed integer type

Arguments

Datatype_id

An integer representing the datatype’s identifier to be queried.

Keywords

None.

Version History

Introduced 5.6

See Also

H5T_GET_ORDER, H5T_GET_PAD, H5T_GET_PRECISION

H5T_GET_SIZE

The H5T_GET_SIZE function returns the size of a datatype in bytes.

Syntax

Result = H5T_GET_SIZE(*Datatype_id*)

Return Value

Returns an integer representing the datatype’s size.

Arguments

Datatype_id

An integer representing the datatype’s identifier to be queried.

Keywords

None.

Version History

Introduced 5.6

See Also

H5T_GET_CLASS, H5T_GET_SUPER

H5T_GET_STRPAD

The H5T_GET_STRPAD function returns the padding method for a string datatype.

Syntax

Result = H5T_GET_STRPAD(*Datatype_id*)

Return Value

Returns a string containing the padding method. Possible values are:

- 'NULLTERM' — Null terminate (like C)
- 'NULLPAD' — Pad with zeroes
- 'SPACEPAD' — Pad with spaces (like FORTRAN)
- 'ERROR'

Arguments

Datatype_id

An integer representing the datatype's identifier to be queried.

Keywords

None.

Version History

Introduced 5.6

See Also

H5T_GET_CSET, H5T_GET_SIZE

H5T_GET_SUPER

The H5T_GET_SUPER function returns the base datatype from which a datatype is derived.

Syntax

Result = H5T_GET_SUPER(*Datatype_id*)

Return Value

Returns an integer representing the base datatype's identifier number. This identifier can be released with the H5T_CLOSE.

Arguments

Datatype_id

An integer representing the datatype's identifier to be queried.

Keywords

None.

Version History

Introduced 5.6

See Also

H5T_GET_CLASS, H5T_GET_SIZE

H5T_IDLTYPE

The H5T_IDLTYPE function returns the IDL type code corresponding to a datatype.

Note

This function is not part of the standard HDF5 interface, but is provided as a programming convenience.

Syntax

Result = H5T_IDLTYPE(*Datatype_id*
[, ARRAY_DIMENSIONS=*variable*][, STRUCTURE=*variable*])

Return Value

The *Result* is an integer giving the IDL type code.

Note

For a list of IDL type codes and their definitions, see "IDL Type Codes" in the *IDL Reference Guide* manual under the SIZE function.

Arguments

Datatype_id

An integer giving the datatype identifier for which to return the IDL type code.

Keywords

ARRAY_DIMENSIONS

Set this keyword to a named variable in which to return a vector containing the array dimensions, if the datatype is an array. If the datatype is not an array, then a scalar value of 0 is returned.

STRUCTURE

Set this keyword to a named variable in which to return the IDL structure definition, if the datatype is a compound datatype. If the datatype is not compound, then a scalar value of 0 is returned.

Version History

Introduced 5.6

See Also

H5T_MEMTYPE

H5T_MEMTYPE

The H5T_MEMTYPE function returns the native memory datatype corresponding to a file datatype.

Note

This function is not part of the standard HDF5 interface, but is provided as a programming convenience.

Syntax

Result = H5T_MEMTYPE(*Datatype_id*)

Return Value

The *Result* is an integer giving the datatype identifier. If the file datatype is not immutable, then the memory datatype identifier should be closed using H5T_CLOSE.

Note

For a list of IDL type codes and their definitions, see "IDL Type Codes" in the *IDL Reference Guide* manual under the SIZE function.

Arguments

Datatype_id

An integer giving the file datatype identifier for which to return the memory datatype.

Keywords

None.

Version History

Introduced 5.6

See Also

H5T_IDLTYPE

H5T_OPEN

The H5T_OPEN function opens a named datatype.

Syntax

Result = H5T_OPEN(*Loc_id*, *Name*)

Return Value

Returns an integer representing the datatype's identifier number. This identifier can be released with the H5T_CLOSE.

Arguments

Loc_id

An integer representing the identifier of the file or group containing the datatype.

Name

A string representing the name of the datatype to be accessed.

Keywords

None.

Version History

Introduced 5.6

See Also

H5T_CLOSE

LA_CHOLDC

The LA_CHOLDC procedure computes the Cholesky factorization of an *n*-by-*n* symmetric (or Hermitian) positive-definite array as:

If A is real: $A = U^T U \text{ or } A = L L^T$

If A is complex: $A = U^H U \text{ or } A = L L^H$

where *U* and *L* are upper and lower triangular arrays. The *T* represents the transpose while *H* represents the Hermitian, or transpose complex conjugate.

LA_CHOLDC is based on the following LAPACK routines:

Output Type	LAPACK Routine
Float	spotrf
Double	dpotrf
Complex	cpotrf
Double complex	zpotrf

Table 3-7: LAPACK Routine Basis for LA_CHOLDC

For more details, see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

LA_CHOLDC, *Array* [, /DOUBLE] [, STATUS=*variable*] [, /UPPER]

Arguments

Array

A named variable containing the real or complex array to be factorized. Only the lower triangular portion of *Array* is used (or upper if the UPPER keyword is set). This procedure returns *Array* as a lower triangular array from the Cholesky decomposition (upper triangular if the UPPER keyword is set).

Keywords

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set DOUBLE = 0 to use single-precision for computations and to return a single-precision (real or complex) result. The default is /DOUBLE if *Array* is double precision, otherwise the default is DOUBLE = 0.

STATUS

Set this keyword to a named variable that will contain the status of the computation. Possible values are:

- STATUS = 0: The computation was successful.
- STATUS > 0: The array is not positive definite and the factorization could not be completed. The STATUS value specifies the order of the leading minor which is not positive definite.

Note

If STATUS is not specified, any error messages will output to the screen.

UPPER

If this keyword is set, then only the upper triangular portion of *Array* is used, and the upper triangular array is returned. The default is to use the lower triangular portion and to return the lower triangular array.

Examples

The following example program computes the Cholesky decomposition of a given symmetric positive-definite array:

```
PRO ExLA_CHOLDC
; Create a symmetric positive-definite array.
n = 10
seed = 12321
array = RANDOMU(seed, n, n)
array = array ## TRANSPOSE(Array)

; Compute the Cholesky decomposition.
lower = array    ; make a copy
LA_CHOLDC, lower
```

```
; Zero out the upper triangular portion.
for i = 0,n - 2 Do lower[i+1:*,i] = 0

; Reconstruct the array and check the difference
arecon = lower ## TRANSPOSE(lower)
PRINT, 'LA_CHOLDC Error:', MAX(ABS(arecon - array))
END
```

When this program is compiled and run, IDL prints:

```
LA_CHOLDC Error:
4.76837e-007
```

Version History

Introduced 5.6

See Also

CHOLDC, LA_CHOLMPROVE, LA_CHOLSOL

LA_CHOLMPROVE

The LA_CHOLMPROVE function uses Cholesky factorization to improve the solution to a system of linear equations, $AX = B$ (where A is symmetric or Hermitian), and provides optional error bounds and backward error estimates.

The LA_CHOLMPROVE function may also be used to improve the solutions for multiple systems of linear equations, with each column of B representing a different set of equations. In this case, the result is a k-by-n array where each of the k columns represents the improved solution vector for that set of equations.

LA_CHOLMPROVE is based on the following LAPACK routines:

Output Type	LAPACK Routine
Float	sporfs
Double	dporfs
Complex	cporfs
Double complex	zporfs

Table 3-8: LAPACK Routine Basis for LA_CHOLMPROVE

For more details, see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

Result = LA_CHOLMPROVE(*Array*, *Achol*, *B*, *X*
[, BACKWARD_ERROR=*variable*] [, /DOUBLE]
[, FORWARD_ERROR=*variable*] [, /UPPER])

Return Value

The result is an n-element vector or k-by-n array.

Arguments

Array

The original n-by-n array of the linear system $AX = B$.

Achol

The *n*-by-*n* Cholesky factorization of *Array*, created by the LA_CHOLDC procedure.

B

An *n*-element input vector containing the right-hand side of the linear system, or a *k*-by-*n* array, where each of the *k* columns represents a different linear system.

X

An *n*-element input vector, or a *k*-by-*n* array, containing the approximate solutions to the linear system, created by the LA_CHOLSOL function.

Keywords

BACKWARD_ERROR

Set this keyword to a named variable that will contain the relative backward error estimate for each linear system. If *B* is a vector containing a single linear system, then BACKWARD_ERROR will be a scalar. If *B* is an array containing *k* linear systems, then BACKWARD_ERROR will be a *k*-element vector. The backward error is the smallest relative change in any element of *A* or *B* that makes *X* an exact solution.

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set DOUBLE = 0 to use single-precision for computations and to return a single-precision (real or complex) result. The default is /DOUBLE if *Array* is double precision, otherwise the default is DOUBLE = 0.

FORWARD_ERROR

Set this keyword to a named variable that will contain the estimated forward error bound for each linear system. If *B* is a vector containing a single linear system, then FORWARD_ERROR will be a scalar. If *B* is an array containing *k* linear systems, then FORWARD_ERROR will be a *k*-element vector. For each linear system, if *Xtrue* is the true solution corresponding to *X*, then the forward error is an estimated upper bound for the magnitude of the largest element in (*X* - *Xtrue*) divided by the magnitude of the largest element in *X*.

UPPER

Set this keyword if *A* contains the upper (rather than lower) triangular array.

Note

If the UPPER keyword is set in LA_CHOLDC and LA_CHOLSOL then the UPPER keyword must also be set in LA_CHOLMPROVE.

Examples

The following example program computes an improved solution to a set of 10 equations:

```
PRO ExLA_CHOLMPROVE
; Create a symmetric positive-definite array.
n = 10
seed = 12321
a = RANDOMU(seed, n, n, /DOUBLE)
a = a ## TRANSPOSE(a)

; Create the right-hand side vector b:
b = RANDOMU(seed, n, /DOUBLE)

; Compute the Cholesky decomposition.
achol = a    ; make a copy
LA_CHOLDC, achol

; Compute the first approximation to the solution:
x = LA_CHOLSOL(achol, b)

; Improve the solution and print the error estimate:
xmprove = LA_CHOLMPROVE(a, achol, b, x, $
     FORWARD_ERROR = fError)
PRINT, 'LA_CHOLMPROVE error:', $
     MAX(ABS(a ## xmprove - b))
PRINT, 'LA_CHOLMPROVE Error Estimate:', fError
END
```

When this program is compiled and run, IDL prints:

```
LA_CHOLMPROVE error:   3.9412917e-15
LA_CHOLMPROVE error estimate:   5.1265892e-12
```

Version History

Introduced 5.6

See Also

LA_CHOLDC, LA_CHOLSOL

LA_CHOLSOL

The LA_CHOLSOL function is used in conjunction with the LA_CHOLDC to solve a set of *n* linear equations in *n* unknowns, *AX* = *B*, where *A* must be a symmetric (or Hermitian) positive-definite array. The parameter *A* is input not as the original array, but as its Cholesky decomposition, created by the routine LA_CHOLDC.

The LA_CHOLSOL function may also be used to solve for multiple systems of linear equations, with each column of *B* representing a different set of equations. In this case, the result is a *k*-by-*n* array where each of the *k* columns represents the solution vector for that set of equations.

LA_CHOLSOL is based on the following LAPACK routines:

Output Type	LAPACK Routine
Float	spotrs
Double	dpotrs
Complex	cpotrs
Double complex	zpotrs

Table 3-9: LAPACK Routine Basis for LA_CHOLSOL

For more details, see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

Result = LA_CHOLSOL(*A*, *B* [, /DOUBLE] [, /UPPER])

Return Value

The result is an *n*-element vector or *k*-by-*n* array.

Arguments

A

The *n*-by-*n* Cholesky factorization of an array, created by the LA_CHOLDC procedure.

B

An *n*-element input vector containing the right-hand side of the linear system, or a *k*-by-*n* array, where each of the *k* columns represents a different linear system.

Keywords

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set DOUBLE = 0 to use single-precision for computations and to return a single-precision (real or complex) result. The default is /DOUBLE if *A* is double precision, otherwise the default is DOUBLE = 0.

UPPER

Set this keyword if *A* contains the upper triangular array, rather than the lower triangular array.

Note

If the UPPER keyword is set in the LA_CHOLDC then the UPPER keyword must also be set in LA_CHOLSOL.

Examples

Given the following system of equations:

$$6u + 15v + 55w = 9.5$$

$$15u + 55v + 225w = 50$$

$$55u + 225v + 979w = 237$$

The solution can be derived by using the following program:

```
PRO ExLA_CHOLSOL
; Define the coefficient array:
a =  [[6.0, 15.0, 55.0], $
     [15.0, 55.0, 225.0], $
     [55.0, 225.0, 979.0]]

; Define the right-hand side vector b:
b = [9.5, 50.0, 237.0]

; Compute the Cholesky decomposition of a:
achol = a    ; make a copy
```

```
LA_CHOLDC, achol

; Compute and print the solution:
x = LA_CHOLSOL(achol, b)
PRINT, 'LA_CHOLSOL solution:', x
END
```

When this program is compiled and run, IDL prints:

```
LA_CHOLSOL Solution:
-0.499999     -1.00000     0.500000
```

The exact solution vector is [-0.5, -1.0, 0.5].

Version History

Introduced 5.6

See Also

CHOLSOL, LA_CHOLDC, LA_CHOLMPROVE

LA_DETERM

The LA_DETERM function uses LU decomposition to compute the determinant of a square array.

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

Syntax

Result = LA_DETERM(*A* [, /CHECK] [, /DOUBLE] [, ZERO=*value*])

Return Value

The result is a scalar of the same type as the input array.

Arguments

A

An *n*-by-*n* real or complex array.

Keywords

CHECK

Set this keyword to check *A* for any singularities. The determinant of a singular array is returned as zero if this keyword is set. Run-time errors may result if *A* is singular and this keyword is not set.

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set DOUBLE = 0 to use single-precision for computations and to return a single-precision (real or complex) result. The default is /DOUBLE if *A* is double precision, otherwise the default is DOUBLE = 0.

ZERO

Use this keyword to set the absolute value of the floating-point zero. A floating-point zero on the main diagonal of a triangular array results in a zero determinant. For single-precision inputs, the default value is 1.0 x 10^{-6}. For double-precision inputs, the default value is 1.0 x 10^{-12}. Setting this keyword to a value less than the default may improve the precision of the result.

Examples

The following program computes the determinant of a square array:

```
PRO ExLA_DETERM
; Create a square array.
array =[[1d, 2, 1], $
     [4, 10, 15], $
     [3, 7, 1]]
; Compute the determinant.
adeterm = LA_DETERM(array)
PRINT, 'LA_DETERM:', adeterm
END
```

When this program is compiled and run, IDL prints:

```
A_DETERM:
-15.000000
```

Version History

Introduced 5.6

See Also

DETERM, LA_LUDC

LA_EIGENPROBLEM

The LA_EIGENPROBLEM function uses the QR algorithm to compute all eigenvalues λ and eigenvectors $v \neq 0$ of an n-by-n real nonsymmetric or complex non-Hermitian array A, for the eigenproblem $Av = \lambda v$. The routine can also compute the left eigenvectors $u \neq 0$, which satisfy $u^H A = \lambda u^H$.

LA_EIGENPROBLEM may also be used for the generalized eigenproblem:

$$Av = \lambda Bv \text{ and } u^H A = \lambda u^H B$$

where A and B are square arrays, v are the right eigenvectors, and u are the left eigenvectors.

LA_EIGENPROBLEM is based on the following LAPACK routines:

Output Type	Standard LAPACK Routine	Generalized LAPACK Routine
Float	sgeevx	sggevx
Double	dgeevx	dggevx
Complex	cgeevx	cggevx
Double complex	zgeevx	zggevx

Table 3-10: LAPACK Routine Basis for LA_EIGENPROBLEM

For details see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

Result = LA_EIGENPROBLEM(*A* [, *B*] [, ALPHA=*variable*] [, BALANCE=*value*] [, BETA=*variable*] [, /DOUBLE] [, EIGENVECTORS=*variable*] [, LEFT_EIGENVECTORS=*variable*] [, NORM_BALANCE = *variable*] [, PERMUTE_RESULT=*variable*] [, SCALE_RESULT=*variable*] [, RCOND_VALUE=*variable*] [, RCOND_VECTOR=*variable*] [, STATUS=*variable*])

Return Value

The result is a complex n-element vector containing the eigenvalues.

Arguments

A

The real or complex array for which to compute eigenvalues and eigenvectors.

B

An optional real or complex *n*-by-*n* array used for the generalized eigenproblem. The elements of *B* are converted to the same type as *A* before computation.

Keywords

ALPHA

For the generalized eigenproblem with the *B* argument, set this keyword to a named variable in which the numerator of the eigenvalues will be returned as a complex *n* -element vector. For the standard eigenproblem this keyword is ignored.

Tip

The ALPHA and BETA values are useful for eigenvalues which underflow or overflow. In this case the eigenvalue problem may be rewritten as $\alpha Av = \beta Bv$.

BALANCE

Set this keyword to one of the following values:

- BALANCE = 0: No balancing is applied to *A*.
- BALANCE = 1: Both permutation and scale balancing are performed.
- BALANCE = 2: Permutations are performed to make the array more nearly upper triangular.
- BALANCE = 3: Diagonally scale the array to make the columns and rows more equal in norm.

The default is BALANCE = 1, which performs both permutation and scaling balances. Balancing a nonsymmetric (or non-Hermitian) array is recommended to reduce the sensitivity of eigenvalues to rounding errors.

BETA

For the generalized eigenproblem with the *B* argument, set this keyword to a named variable in which the denominator of the eigenvalues will be returned as a real or complex *n*-element vector. For the standard eigenproblem this keyword is ignored.

Tip

The ALPHA and BETA values are useful for eigenvalues which underflow or overflow. In this case, the eigenvalue problem may be rewritten as $\alpha Av = \beta Bv$.

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set DOUBLE = 0 to use single-precision for computations and to return a single-precision (real or complex) result. The default is /DOUBLE if *A* is double precision, otherwise the default is DOUBLE = 0.

EIGENVECTORS

Set this keyword to a named variable in which the eigenvectors will be returned as a set of row vectors. If this variable is omitted then eigenvectors will not be computed unless the RCOND_VALUE or RCOND_VECTOR keywords are present.

Note

For the standard eigenproblem the eigenvectors are normalized and rotated to have norm 1 and largest component real. For the generalized eigenproblem the eigenvectors are normalized so that the largest component has abs(real) + abs(imaginary) = 1.

LEFT_EIGENVECTORS

Set this keyword to a named variable in which the left eigenvectors will be returned as a set of row vectors. If this variable is omitted then left eigenvectors will not be computed unless the RCOND_VALUE or RCOND_VECTOR keywords are present.

Note

Note - For the standard eigenproblem the eigenvectors are normalized and rotated to have norm 1 and largest component real. For the generalized eigenproblem the eigenvectors are normalized so that the largest component has abs(real) + abs(imaginary) = 1.

NORM_BALANCE

Set this keyword to a named variable in which the one-norm of the balanced matrix will be returned. The one-norm is defined as the maximum value of the sum of absolute values of the columns. For the standard eigenproblem, this will be returned as a scalar value; for the generalized eigenproblem this will be returned as a two-element vector containing the *A* and *B* norms.

PERMUTE_RESULT

Set this keyword to a named variable in which the result for permutation balancing will be returned as a two-element vector [*ilo*, *ihi*]. If permute balancing is not done then the values will be *ilo* = *1* and *ihi* = *n*.

RCOND_VALUE

Set this keyword to a named variable in which the reciprocal condition numbers for the eigenvalues will be returned as an *n*-element vector. If RCOND_VALUE is present then left and right eigenvectors must be computed.

RCOND_VECTOR

Set this keyword to a named variable in which the reciprocal condition numbers for the eigenvectors will be returned as an *n*-element vector. If RCOND_VECTOR is present then left and right eigenvectors must be computed.

SCALE_RESULT

Set this keyword to a named variable in which the results for permute and scale balancing will be returned. For the standard eigenproblem, this will be returned as an *n*-element vector. For the generalized eigenproblem, this will be returned as a *n*-by-2 array with the first row containing the permute and scale factors for the left side of *A* and *B* and the second row containing the factors for the right side of *A* and *B*.

STATUS

Set this keyword to a named variable that will contain the status of the computation. Possible values are:

- STATUS = 0: The computation was successful.
- STATUS > 0: The QR algorithm failed to compute all eigenvalues; no eigenvectors or condition numbers were computed. The STATUS value indicates that eigenvalues *ilo:STATUS* (starting at index 1) did not converge; all other eigenvalues converged.

Note

If STATUS is not specified, any error messages will be output to the screen.

Examples

Find the eigenvalues and eigenvectors for an array using the following program:

```
PRO ExLA_EIGENPROBLEM
; Create a random array:
n = 4
seed = 12321
array = RANDOMN(seed, n, n)

; Compute all eigenvalues and eigenvectors:
eigenvalues = LA_EIGENPROBLEM(array, $
     EIGENVECTORS = eigenvectors)
PRINT, 'LA_EIGENPROBLEM Eigenvalues:'
PRINT, eigenvalues

; Check the results using the eigenvalue equation:
maxErr = 0d
FOR i = 0, n - 1 DO BEGIN
     ; A*z = lambda*z
     alhs = array ## eigenvectors[*,i]
     arhs = eigenvalues[i]*eigenvectors[*,i]
     maxErr = maxErr > MAX(ABS(alhs - arhs))
ENDFOR
PRINT, 'LA_EIGENPROBLEM Error:', maxErr

; Now try the generalized eigenproblem:
b = IDENTITY(n) + 0.01*RANDOMN(seed, n, n)
eigenvalues = LA_EIGENPROBLEM(Array, B)
PRINT, 'LA_EIGENPROBLEM Generalized Eigenvalues:'
PRINT, EIGENVALUES
END
```

When this program is compiled and run, IDL prints:

```
LA_EIGENPROBLEM eigenvalues:
(    -0.593459,     0.566318)(    -0.593459,    -0.566318)
(      1.06216,      0.00000)(      1.61286,      0.00000)
LA_EIGENPROBLEM error:   4.0978193e-07
LA_EIGENPROBLEM generalized eigenvalues:
(    -0.574766,     0.567452)(    -0.574766,    -0.567452)
(      1.57980,      0.00000)(      1.08711,      0.00000)
```

Version History

Introduced 5.6

See Also

LA_EIGENVEC, LA_ELMHES, LA_HQR

LA_EIGENQL

The LA_EIGENQL function computes selected eigenvalues λ and eigenvectors $z \neq 0$ of an n-by-n real symmetric or complex Hermitian array A, for the eigenproblem $Az = \lambda z$.

LA_EIGENQL may also be used for the generalized symmetric eigenproblems:

$$Az = \lambda Bz \ \text{ or } \ ABz = \lambda z \ \text{ or } \ BAz = \lambda z$$

where A and B are symmetric (or Hermitian) and B is positive definite.

LA_EIGENQL is based on the following LAPACK routines:

Output Type	Standard Eigenproblem	Generalized
Float	ssyevx, ssyevr, ssyevd	ssygvx, ssygvd
Double	dsyevx, dsyevr, dsyevd	dsygvx, dsygvd
Complex	cheevx, cheevr, cheevd	chegvx, chegvd
Double complex	zheevx, zheevr, zheevd	zhegvx, zhegvd

Table 3-11: LAPACK Routine Basis for LA_EIGENQL

For details see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

Result = LA_EIGENQL(*A* [, B] [, /DOUBLE] [, EIGENVECTORS=*variable*] [, FAILED=*variable*] [, GENERALIZED=*value*] [, METHOD=*value*] [, RANGE=*vector*] [, SEARCH_RANGE=*vector*] [, STATUS=*variable*] [, TOLERANCE=*value*])

Return Value

The result is a real vector containing the eigenvalues in ascending order.

Arguments

A

The real or complex *n*-by-*n* array for which to compute eigenvalues and eigenvectors. A must be symmetric (or Hermitian).

B

An optional real or complex *n*-by-*n* array used for the generalized eigenproblem. *B* must be symmetric (or Hermitian) and positive definite. The elements of *B* are converted to the same type as *A* before computation.

Keywords

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set DOUBLE = 0 to use single-precision for computations and to return a single-precision (real or complex) result. The default is /DOUBLE if *A* is double precision, otherwise the default is DOUBLE = 0.

EIGENVECTORS

Set this keyword to a named variable in which the eigenvectors will be returned as a set of row vectors. If this variable is omitted then eigenvectors will not be computed. All eigenvectors will be returned unless the RANGE or SEARCH_RANGE keywords are used to restrict the eigenvalue range.

FAILED

Set this keyword to a named variable in which to return the indices of eigenvectors that did not converge. This keyword is only available for METHOD = 0, and will be ignored for other methods.

Note

Index numbers within FAILED start at 1.

GENERALIZED

For the generalized eigenproblem with the optional B argument, set this keyword to indicate which problem to solve. Possible values are:

- GENERALIZED = 0 (the default): Solve $Az = \lambda Bz$.

- GENERALIZED = 1: Solve $ABz = \lambda z$.
- GENERALIZED = 2: Solve $BAz = \lambda z$.

This keyword is ignored if argument *B* is not present.

METHOD

Set this keyword to indicate which computation method to use. Possible values are:

- METHOD = 0 (the default): Use tridiagonal decomposition to compute some or all of the eigenvalues and (optionally) eigenvectors.
- METHOD = 1: Use the Relatively Robust Representation (RRR) algorithm to compute some or all of the eigenvalues and (optionally) eigenvectors. This method is unavailable for the generalized eigenproblem with the optional *B* argument, and will default to METHOD = 0.

Note

The RRR method may produce *NaN* and *Infinity* floating-point exception messages during normal execution.

- METHOD = 2: Use a divide-and-conquer algorithm to compute all of the eigenvalues and (optionally) all eigenvectors. This method is available for either the standard or generalized eigenproblems. For METHOD = 2 the RANGE, SEARCH_RANGE, and TOLERANCE keywords are ignored, and all eigenvalues are returned.

RANGE

Set this keyword to a two-element vector containing the indices of the smallest and largest eigenvalues to be returned. The default is [0, *n*-1], which returns all eigenvalues and eigenvectors. This keyword is ignored for METHOD = 2.

SEARCH_RANGE

Set this keyword to a two-element floating-point vector containing the lower and upper bounds of the interval to be searched for eigenvalues. The default is to return all eigenvalues and eigenvectors. This keyword is ignored for METHOD = 2. If both RANGE and SEARCH_RANGE are specified, only the SEARCH_RANGE values are used.

Note

If the search range does not contain any eigenvalues, then Result, EIGENVECTORS, and FAILED will each be set to a scalar zero.

STATUS

Set this keyword to a named variable that will contain the status of the computation. In all cases STATUS = 0 indicates successful computation. For the standard eigenproblem, possible nonzero values are:

- METHOD = 0, STATUS > 0: STATUS eigenvectors failed to converge. The FAILED keyword contains the indices of the eigenvectors that did not converge.
- METHOD = 1, STATUS < 0 or STATUS > 0: An internal error occurred during the computation.
- METHOD = 2, STATUS > 0: STATUS off-diagonal elements of an intermediate tridiagonal matrix did not converge to zero.

For the generalized eigenproblem, possible nonzero values are:

- METHOD = 0, 0 < STATUS ≤ *n*: STATUS eigenvectors failed to converge. The FAILED keyword contains the indices of the eigenvectors that did not converge.
- METHOD = 0, STATUS > *n*: The factorization of *B* could not be completed and the computation failed. The value of (STATUS - *n*) specifies the order of the leading minor of *B* which is not positive definite.
- METHOD = 2, 0 < STATUS ≤ *n*: STATUS off-diagonal elements of an intermediate tridiagonal matrix did not converge to zero.
- METHOD = 2, STATUS > *n*: The factorization of *B* could not be completed and the computation failed. The value of (STATUS - *n*) specifies the order of the leading minor of *B* which is not positive definite.

Note

If STATUS is not specified, any error messages will be output to the screen.

TOLERANCE

Set this keyword to a scalar giving the absolute error tolerance for the eigenvalues and eigenvectors. For the most accurate eigenvalues, TOLERANCE should be set to 2**XMIN*, where *XMIN* is the magnitude of the smallest usable floating-point value. For METHOD = 0, if TOLERANCE is less than or equal to zero, or is unspecified, then a tolerance value of $EPS*||T||_1$ will be used, where T is the tridiagonal matrix obtained from A. For METHOD = 1, if TOLERANCE is less than or equal to $N*EPS*||T||_1$, or is unspecified, then a tolerance value of $N*EPS*||T||_1$ will be used.

For values of *EPS* and *XMIN*, see the MACHAR. This keyword is ignored for METHOD = 2.

Tip

If the LA_EIGENQL routine fails to converge, try setting the TOLERANCE to a larger value.

Examples

Find the eigenvalues and eigenvectors for a symmetric array using the following program:

```
PRO ExLA_EIGENQL
; Create a random symmetric array:
n = 10
seed = 12321
array = RANDOMN(seed, n, n)
array = array + TRANSPOSE(array)

; Compute all eigenvalues and eigenvectors:
eigenvalues = LA_EIGENQL(array, $
     EIGENVECTORS=eigenvectors)

; Check the results using the eigenvalue equation:
maxErr = 0d
FOR i=0,n-1 DO BEGIN
     ; a*z = lambda*z
     alhs = array ## eigenvectors[*,i]
     arhs = eigenvalues[i]*eigenvectors[*,i]
     maxErr = maxErr > MAX(ABS(alhs - arhs))
ENDFOR
PRINT, 'LA_EIGENQL error:', maxErr

; Compute the three largest eigenvalues:
eigenvalues = LA_EIGENQL(array, $
     EIGENVECTORS = eigenvectors, $
     RANGE = [n-3,n-1])
PRINT, 'LA_EIGENQL eigenvalues:', eigenvalues

; Now try the generalized eigenproblem:
b = IDENTITY(n) + 0.01*RANDOMN(seed,n,n)
; Make B symmetric and positive definite:
b = b ## TRANSPOSE(b)
```

```
; Compute the three largest generalized eigenvalues:
eigenvalues = LA_EIGENQL(array, b, RANGE=[n-3,n-1])
PRINT, 'LA_EIGENQL Generalized Eigenvalues:'
PRINT, Eigenvalues
END
```

When this program is compiled and run, IDL prints:

```
LA_EIGENQL error:   1.3560057e-06
LA_EIGENQL eigenvalues:      3.82993      4.69785      5.61567
LA_EIGENQL generalized eigenvalues:
3.83750      4.74803      5.57692
```

Version History

Introduced 5.6

See Also

EIGENQL, LA_TRIQL, LA_TRIRED

LA_EIGENVEC

The LA_EIGENVEC function uses the QR algorithm to compute all or some of the eigenvectors $v \neq 0$ of an *n*-by-*n* real nonsymmetric or complex non-Hermitian array *A*, for the eigenproblem $Av = \lambda v$. The routine can also compute the left eigenvectors $u \neq 0$, which satisfy $u^H A = \lambda u^H$.

Note

The left and right eigenvectors returned by LA_EIGENVEC are normalized to norm 1. Unlike the LA_EIGENPROBLEM, they are not rotated to have largest component real. Therefore, you may notice slight differences in results between LA_EIGENVEC and LA_EIGENPROBLEM.

LA_EIGENVEC is based on the following LAPACK routines:

Output Type	Eigenvectors	Condition Numbers	Undo Balancing
Float	strevc	strsna	sgebak
Double	dtrevc	dtrsna	dgebak
Complex	ctrevc	ctrsna	cgebak
Double complex	ztrevc	ztrsna	zgebak

Table 3-12: LAPACK Routine Basis for LA_EIGENVEC

For details see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

Result = LA_EIGENVEC(*T*, *QZ* [, BALANCE=*value*] [, /DOUBLE]
[, EIGENINDEX=*variable*] [, LEFT_EIGENVECTORS=*variable*]
[, PERMUTE_RESULT=[*ilo*, *ihi*]] [, SCALE_RESULT=*vector*]
[, RCOND_VALUE=*variable*] [, RCOND_VECTOR=*variable*] [, SELECT=*vector*])

Return Value

The result is a complex array containing the eigenvectors as a set of row vectors.

Arguments

T

The upper quasi-triangular array containing the Schur form, created by LA_HQR.

QZ

The array of Schur vectors, created by LA_HQR.

Keywords

BALANCE

If balancing was applied in the call to LA_ELMHES, then set this keyword to the same value that was used, in order to apply the backward balancing transform to the eigenvectors. If BALANCE is not specified, then the default is BALANCE = 1.

Note

If BALANCE is not zero, then both PERMUTE_RESULT and SCALE_RESULT must be supplied.

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set DOUBLE = 0 to use single-precision for computations and to return a single-precision (real or complex) result. The default is /DOUBLE if *T* is double precision, otherwise the default is DOUBLE = 0.

EIGENINDEX

If keyword SELECT is used, then set this keyword to a named variable in which the indices of the eigenvalues that correspond to the selected eigenvectors will be returned. If the SELECT keyword is not used then EIGENINDEX will be set to LINDGEN(*n*).

Tip

This keyword is most useful for real input arrays when the SELECT keyword is present. In this case, a value of SELECT[*j*] equal to 1 may produce two eigenvectors if the eigenvalue is part of a complex-conjugate pair.

LEFT_EIGENVECTORS

Set this keyword to a named variable in which the left eigenvectors will be returned as a set of row vectors. If this variable is omitted then left eigenvectors will not be computed unless the RCOND_VALUE keyword is present.

PERMUTE_RESULT

Set this keyword to a two-element vector containing the [*ilo*, *ihi*] permutation results from the LA_ELMHES procedure. This keyword must be present if BALANCE = 1 or BALANCE = 2.

RCOND_VALUE

Set this keyword to a named variable in which the reciprocal condition numbers for the eigenvalues will be returned as an *n*-element vector. If RCOND_VALUE is present then left and right eigenvectors must be computed.

RCOND_VECTOR

Set this keyword to a named variable in which the reciprocal condition numbers for the eigenvectors will be returned as an *n*-element vector.

SCALE_RESULT

Set this keyword to an n-element vector containing the permute and scale balancing results from the LA_ELMHES procedure. This keyword must be present if BALANCE is not zero.

SELECT

Set this keyword to an *n*-element vector of zeroes or ones that indicates which eigenvectors to compute. There are two cases:

- The original array was real: If the *j*-th eigenvalue (as created by LA_HQR) is real, then if SELECT[*j*] is set to 1, then the *j*-th eigenvector will be computed. If the *j*-th and (*j*+1) eigenvalues form a complex-conjugate pair, then if either SELECT[*j*] or SELECT[*j*+1] is set to 1, then the complex-conjugate pair of *j*-th and (*j*+1) eigenvectors will be computed.
- The original array was complex: If SELECT[*j*] is set to 1, then the *j*-th eigenvector will be computed.

If SELECT is omitted then all eigenvectors are returned.

Examples

Compute the eigenvalues and selected eigenvectors of a random array using the following program:

```
PRO ExLA_EIGENVEC
; Create a random array:
n = 10
seed = 12321
array = RANDOMN(seed, n, n)

     ; Reduce to upper Hessenberg and compute Q:
     H = LA_ELMHES(array, q, $
     PERMUTE_RESULT = permute, SCALE_RESULT = scale)

; Compute eigenvalues, T, and QZ arrays:
eigenvalues = LA_HQR(h, q, PERMUTE_RESULT = permute)

; Compute eigenvectors corresponding to
; the first 3 eigenvalues.
select = [1, 1, 1, REPLICATE(0, n - 3)]
eigenvectors = LA_EIGENVEC(H, Q, $
     EIGENINDEX = eigenindex, $
     PERMUTE_RESULT = permute, SCALE_RESULT = scale, $
     SELECT = select)

PRINT, 'LA_EIGENVEC eigenvalues:'
PRINT, eigenvalues[eigenindex]
END
```

When this program is compiled and run, IDL prints:

```
LA_EIGENVEC eigenvalues:
(-0.278633, 2.55055) ( -0.278633, -2.55055)
(2.31208,     0.000000)
```

Version History

Introduced 5.6

See Also

EIGENVEC, LA_ELMHES, LA_HQR

LA_ELMHES

The LA_ELMHES function reduces a real nonsymmetric or complex non-Hermitian array to upper Hessenberg form *H*. If the array is real then the decomposition is $A = Q H Q^T$, where *Q* is orthogonal. If the array is complex Hermitian then the decomposition is $A = Q H Q^H$, where *Q* is unitary. The superscript T represents the transpose while superscript *H* represents the Hermitian, or transpose complex conjugate.

LA_ELMHES is based on the following LAPACK routines:

Output Type	Balance & Reduce	Norm	Optional Q
Float	sgebal, sgehrd	slange	sorghr
Double	dgebal, dgehrd	dlange	dorghr
Complex	cgebal, cgehrd	clange	cunghr
Double complex	zgebal, zgehrd	zlange	zunghr

Table 3-13: LAPACK Routine Basis for LA_ELMHES

For details see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

Result = LA_ELMHES(*Array* [, Q] [, BALANCE=*value*] [, /DOUBLE]
[, NORM_BALANCE=*variable*] [, PERMUTE_RESULT=*variable*]
[, SCALE_RESULT=*variable*])

Return Value

The result is an array of the same type as *A* containing the upper Hessenberg form. The Hessenberg array is stored in the upper triangle and the first subdiagonal. Elements below the subdiagonal should be ignored but are not automatically set to zero.

Arguments

Array

The *n*-by-*n* real or complex array to reduce to upper Hessenberg form.

Q

Set this optional argument to a named variable in which the array *Q* will be returned. The *Q* argument may then be input into LA_HQR to compute the Schur vectors.

Keywords

BALANCE

Set this keyword to one of the following values:

- BALANCE = 0: No balancing is applied to *Array*.
- BALANCE = 1: Both permutation and scale balancing are performed.
- BALANCE = 2: Permutations are performed to make the array more nearly upper triangular.
- BALANCE = 3: Diagonally scale the array to make the columns and rows more equal in norm.

The default is BALANCE = 1, which performs both permutation and scaling balances. Balancing a nonsymmetric array is recommended to reduce the sensitivity of eigenvalues to rounding errors.

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set DOUBLE = 0 to use single-precision for computations and to return a single-precision (real or complex) result. The default is /DOUBLE if *Array* is double precision, otherwise the default is DOUBLE = 0.

NORM_BALANCE

Set this keyword to a named variable in which the one-norm of the balanced matrix will be returned. The one-norm is defined as the maximum value of the sum of absolute values of the columns.

PERMUTE_RESULT

Set this keyword to a named variable in which the result for permutation balancing will be returned as a two-element vector [*ilo*, *ihi*]. If permute balancing is not done then the values will be *ilo* = *1* and *ihi* = *n*.

SCALE_RESULT

Set this keyword to a named variable in which the result for permute and scale balancing will be returned as an *n*-element vector.

Examples

See LA_EIGENVEC for an example of using this procedure.

Version History

Introduced 5.6

See Also

ELMHES, LA_HQR

LA_GM_LINEAR_MODEL

The LA_GM_LINEAR_MODEL function is used to solve a general Gauss-Markov linear model problem:

$$\text{minimize}_x \; ||y||_2 \text{ with constraint } d = Ax + By$$

where A is an m-column by n-row array, B is a p-column by n-row array, and d is an n-element input vector with $m \leq n \leq m+p$.

The following items should be noted:

- If A has full column rank m and the array (A B) has full row rank n, then there is a unique solution x and a minimal 2-norm solution y.
- If B is square and nonsingular then the problem is equivalent to a weighted linear least-squares problem, $\text{minimize}_x \; ||B^{-1}(Ax - d)||_2$.
- If B is the identity matrix then the problem reduces to the ordinary linear least-squares problem, $\text{minimize}_x \; ||Ax - d||_2$.

LA_ GM_LINEAR_MODEL is based on the following LAPACK routines:

Output Type	LAPACK Routine
Float	sggglm
Double	dggglm
Complex	cggglm
Double complex	zggglm

Table 3-14: LAPACK Routine Basis for LA_GM_LINEAR_MODEL

For details see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

Result = LA_GM_LINEAR_MODEL(*A*, *B*, *D*, *Y* [, /DOUBLE])

Return Value

The result (*x*) is an *m*-element vector whose type is identical to *A*.

Arguments

A

The *m*-by-*n* array used in the constraint equation.

B

The *p*-by-*n* array used in the constraint equation.

D

An *n*-element input vector used in the constraint equation.

Y

Set this argument to a named variable, which will contain the *p*-element output vector.

Keywords

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set DOUBLE = 0 to use single-precision for computations and to return a single-precision (real or complex) result. The default is /DOUBLE if *A* is double precision, otherwise the default is DOUBLE = 0.

Examples

Given the constraint equation $d = Ax + By$, (where *A*, *B*, and *d* are defined in the program below) the following example program solves the general Gauss-Markov problem:

```
PRO ExLA_GM_LINEAR_MODEL
; Define some example coefficient arrays:
a = [[2, 7, 4], $
     [5, 1, 3], $
     [3, 3, 6], $
     [4, 5, 2]]
b = [[-3, 2], $
     [1, 5], $
     [2, 9], $
     [4, 1]]
```

```
; Define a sample left-hand side vector D:
d = [-1, 2, -3, 4]

; Find and print the solution x:
x = LA_GM_LINEAR_MODEL(a, b, d, y)
PRINT, 'LA_GM_LINEAR_MODEL solution:'
PRINT, X
PRINT, 'LA_GM_LINEAR_MODEL 2-norm solution:'
PRINT, Y
END
```

When this program is compiled and run, IDL prints:

```
LA_GM_LINEAR_MODEL solution:
      1.04668     0.350346     -1.28445
LA_GM_LINEAR_MODEL 2-norm solution:
     0.151716    0.0235733
```

Version History

Introduced 5.6

See Also

LA_LEAST_SQUARE_EQUALITY, LA_LEAST_SQUARES

LA_HQR

The LA_HQR function uses the multishift QR algorithm to compute all eigenvalues of an *n*-by-*n* upper Hessenberg array. The LA_ELMHES routine can be used to reduce a real or complex array to upper Hessenberg form suitable for input to this procedure. LA_HQR may also be used to compute the matrices *T* and *QZ* from the Schur decomposition $A = (QZ)\ T\ (QZ)^H$.

LA_HQR is based on the following LAPACK routines:

Output Type	LAPACK Routine
Float	shseqr
Double	dhseqr
Complex	chseqr
Double complex	zhseqr

Table 3-15: LAPACK Routine Basis for LA_HQR

For details see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

Result = LA_HQR(*H* [, Q] [, /DOUBLE] [, PERMUTE_RESULT=[*ilo*, *ihi*]] [, STATUS=*variable*])

Return Value

The result is an *n*-element complex vector.

Arguments

H

An *n*-by-*n* upper Hessenberg array, created by the LA_ELMHES procedure. If argument *Q* is present, then on return *H* is replaced by the Schur form *T*. If argument *Q* is not present then *H* is unchanged.

Q

Set this optional argument to the array *Q* created by the LA_ELMHES procedure. If argument *Q* is present, then on return *Q* is replaced by the Schur vectors *QZ*.

Keywords

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set DOUBLE = 0 to use single-precision for computations and to return a single-precision (real or complex) result. The default is /DOUBLE if *H* is double precision, otherwise the default is DOUBLE = 0.

PERMUTE_RESULT

Set this keyword to a two-element vector containing the [*ilo*, *ihi*] permutation results from the LA_ELMHES procedure. The default is [1, *n*], indicating that permute balancing was not done on *H*.

STATUS

Set this keyword to a named variable that will contain the status of the computation. Possible values are:

- STATUS = 0: The computation was successful.
- STATUS > 0: The algorithm failed to find all eigenvalues in *30**(*ihi* - *ilo* + *1*) iterations. The STATUS value indicates that eigenvalues ilo:STATUS (starting at index *1*) did not converge; all other eigenvalues converged.

Note

If STATUS is not specified, any error messages will output to the screen.

Examples

See LA_EIGENVEC for an example of using this procedure.

Version History

Introduced 5.6

See Also

HQR, LA_EIGENVEC, LA_ELMHES

LA_INVERT

The LA_INVERT function uses LU decomposition to compute the inverse of a square array.

LA_INVERT is based on the following LAPACK routines:

Output Type	LAPACK Routine
Float	sgetrf, sgetri
Double	dgetrf, dgetri
Complex	cgetrf, cgetri
Double complex	zgetrf, zgetri

Table 3-16: LAPACK Routine Basis for LA_INVERT

For more details, see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

Result = LA_INVERT(*A* [, /DOUBLE] [, STATUS=*variable*])

Return Value

The result is an array of the same dimensions as the input array.

Arguments

A

The *n*-by-*n* array to be inverted.

Keywords

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set DOUBLE = 0 to use single-precision for computations and to return a single-precision (real or complex) result. The default is /DOUBLE if *A* is double precision, otherwise the default is DOUBLE = 0.

STATUS

Set this keyword to a named variable that will contain the status of the computation. Possible values are:

- STATUS = 0: The computation was successful.
- STATUS > 0: The array is singular and the inverse could not be computed. The STATUS value specifies which value along the diagonal (starting at one) is zero.

Note

If STATUS is not specified, any error messages will be output to the screen.

Examples

The following program computes the inverse of a square array:

```
PRO ExLA_INVERT
; Create a square array.
array =[[1d, 2, 1], $
     [4, 10, 15], $
     [3, 7, 1]]
; Compute the inverse and check the error.
ainv = LA_INVERT(array)
PRINT, 'LA_INVERT Identity Matrix:'
PRINT, ainv ## array
END
```

When this program is compiled and run, IDL prints:

```
A_INVERT Identity Matrix:
1.0000000   1.7763568e-015  6.6613381e-016
0.00000000  1.0000000       1.2212453e-015
0.00000000  0.00000000      1.0000000
```

Version History

Introduced 5.6

See Also

INVERT, LA_LUDC

LA_LEAST_SQUARE_EQUALITY

The LA_LEAST_SQUARE_EQUALITY function is used to solve the linear least-squares problem:

$$\text{Minimize}_x \; ||Ax - c||_2 \text{ with constraint } Bx = d$$

where A is an n-column by m-row array, B is an n-column by p-row array, c is an m-element input vector, and d is an p-element input vector with $p \leq n \leq m+p$. If B has full row rank p and the array $\begin{pmatrix} A \\ B \end{pmatrix}$ has full column rank n, then a unique solution exists.

LA_ LEAST_SQUARE_EQUALITY is based on the following LAPACK routines:

Output Type	LAPACK Routine
Float	sgglse
Double	dgglse
Complex	cgglse
Double complex	zgglse

Table 3-17: LAPACK Routine Basis for LA_LEAST_SQUARE_EQUALITY

For details see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

Result = LA_LEAST_SQUARE_EQUALITY(*A*, *B*, *C*, *D* [, /DOUBLE] [, RESIDUAL=*variable*])

Return Value

The result (x) is an n-element vector.

Arguments

A

The n-by-m array used in the least-squares minimization.

B

The *n*-by-*p* array used in the equality constraint.

C

An *m*-element input vector containing the right-hand side of the least-squares system.

D

A *p*-element input vector containing the right-hand side of the equality constraint.

Keywords

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set DOUBLE = 0 to use single-precision for computations and to return a single-precision (real or complex) result. The default is /DOUBLE if *A* is double precision, otherwise the default is DOUBLE = 0.

RESIDUAL

Set this keyword to a named variable in which to return a scalar giving the residual sum-of-squares for Result. If $n = m + p$ then RESIDUAL will be zero.

Examples

Given the following system of equations:

```
2t + 5u + 3v + 4w = 9
7t +  u + 3v + 5w = 1
4t + 3u + 6v + 2w = 2
```

with constraints,

```
-3t +  u + 2v + 4w = -4
 2t + 5u + 9v + 1w =  4
```

find the solution using the following program:

```
PRO ExLA_LEAST_SQUARE_EQUALITY
; Define the coefficient array:
a = [[2, 5, 3, 4], $
     [7, 1, 3, 5], $
     [4, 3, 6, 2]]
```

```
; Define the constraint array:
b = [[-3, 1, 2, 4], $
     [2, 5, 9, 1]]

; Define the right-hand side vector c:
c = [9, 1, 2]

; Define the constraint right-hand side d:
d = [-4, 4]

; Find and print the minimum norm solution of a:
x = LA_LEAST_SQUARE_EQUALITY(a, b, c, d)
PRINT, 'LA_LEAST_SQUARE_EQUALITY solution:'
PRINT, x
END
```

When this program is compiled and run, IDL prints:

```
LA_LEAST_SQUARE_EQUALITY solution:
     0.651349      2.72695     -1.14638    -0.620036
```

Version History

Introduced 5.6

See Also

LA_GM_LINEAR_MODEL, LA_LEAST_SQUARES

LA_LEAST_SQUARES

The LA_LEAST_SQUARES function is used to solve the linear least-squares problem:

$$\text{Minimize}_x \|Ax - b\|_2$$

where A is a (possibly rank-deficient) n-column by m-row array, b is an m-element input vector, and x is the n-element solution vector. There are three possible cases:

- If $m \geq n$ and the rank of A is n, then the system is overdetermined and a unique solution may be found, known as the least-squares solution.
- If $m < n$ and the rank of A is m, then the system is under determined and an infinite number of solutions satisfy $Ax - b = 0$. In this case, the solution is found which minimizes $\|x\|_2$, known as the minimum norm solution.
- If A is rank deficient, such that the rank of A is less than MIN(m, n), then the solution is found which minimizes both $||Ax - b||_2$ and $||x||_2$, known as the minimum-norm least-squares solution.

The LA_LEAST_SQUARES function may also be used to solve for multiple systems of least squares, with each column of b representing a different set of equations. In this case, the result is a k-by-n array where each of the k columns represents the solution vector for that set of equations.

LA_ LEAST_SQUARES is based on the following LAPACK routines:

Output Type	LAPACK Routines
Float	sgels, sgelsy, sgelss, sgelsd
Double	dgels, dgelsy, dgelss, dgelsd
Complex	cgels, cgelsy, cgelss, cgelsd
Double complex	zgels, zgelsy, zgelss, zgelsd

Table 3-18: LAPACK Routine Basis for LA_LEAST_SQUARES

For details see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

Result = LA_LEAST_SQUARES(*A*, *B* [, /DOUBLE] [, METHOD=*value*] [, RANK=*variable*] [, RCONDITION=*value*] [, RESIDUAL=*variable*] [, STATUS=*variable*])

Return Value

The result is an *n*-element vector or *k*-by-*n* array.

Arguments

A

The *n*-by-*m* array used in the least-squares system.

B

An *m*-element input vector containing the right-hand side of the linear least-squares system, or a *k*-by-*m* array, where each of the *k* columns represents a different least-squares system.

Keywords

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set DOUBLE = 0 to use single-precision for computations and to return a single-precision (real or complex) result. The default is /DOUBLE if *A* is double precision, otherwise the default is DOUBLE = 0.

METHOD

Set this keyword to indicate which computation method to use. Possible values are:

- METHOD = 0 (the default): Assume that array *A* has full rank equal to min(m, n). If $m \geq n$, find the least-squares solution to the overdetermined system. If $m < n$, find the minimum norm solution to the under determined system. Both cases use QR or LQ factorization of *A*.
- METHOD = 1: Assume that array *A* may be rank deficient; use a complete orthogonal factorization of *A* to find the minimum norm least-squares solution.

- METHOD = 2: Assume that array *A* may be rank deficient; use singular value decomposition (SVD) to find the minimum norm least-squares solution.
- METHOD = 3: Assume that array *A* may be rank deficient; use SVD with a divide-and-conquer algorithm to find the minimum norm least-squares solution. The divide-and-conquer method is faster than regular SVD, but may require more memory.

RANK

Set this keyword to a named variable in which to return the effective rank of *A*. If METHOD = 0 or the array is full rank, then RANK will have the value MIN(*m*, *n*).

RCONDITION

Set this keyword to the reciprocal condition number used as a cutoff value in determining the effective rank of *A*. Arrays with condition numbers larger than 1/RCONDITION are assumed to be rank deficient. If RCONDITION is set to zero or omitted, then array *A* is assumed to be of full rank. This keyword is ignored for METHOD = 0.

RESIDUAL

If $m > n$ and the rank of *A* is *n* (the system is overdetermined), then set this keyword to a named variable in which to return the residual sum-of-squares for *Result*. If *B* is an *m*-element vector then RESIDUAL will be a scalar; if *B* is a *k*-by-*m* array then RESIDUAL will be a *k*-element vector containing the residual sum-of-squares for each system of equations. If $m \leq n$ or *A* is rank deficient (rank < *n*) then the values in RESIDUAL will be zero.

STATUS

Set this keyword to a named variable that will contain the status of the computation. Possible values are:

- STATUS = 0: The computation was successful.
- STATUS > 0: For METHOD=2 or METHOD=3, this indicates that the SVD algorithm failed to converge, and STATUS off-diagonal elements of an intermediate bidiagonal form did not converge to zero. For METHOD=0 or METHOD=1 the STATUS will always be zero.

Examples

Given the following under determined system of equations:

```
2t + 5u + 3v + 4w = 3
7t +  u + 3v + 5w = 1
4t + 3u + 6v + 2w = 6
```

The following program can be used to find the solution:

```
PRO ExLA_LEAST_SQUARES
; Define the coefficient array:
a = [[2, 5, 3, 4], $
     [7, 1, 3, 5], $
     [4, 3, 6, 2]]

; Define the right-hand side vector b:
b = [3, 1, 6]

; Find and print the minimum norm solution of a:
x = LA_LEAST_SQUARES(a, b)
PRINT, 'LA_LEAST_SQUARES solution:', x
END
```

When this program is compiled and run, IDL prints:

```
LA_LEAST_SQUARES solution:
-0.0376844     0.350628     0.986164    -0.409066
```

Version History

Introduced 5.6

See Also

LA_GM_LINEAR_MODEL, LA_LEAST_SQUARE_EQUALITY

LA_LINEAR_EQUATION

The LA_LINEAR_EQUATION function uses LU decomposition to solve a system of linear equations, *AX* = *B*, and provides optional error bounds and backward error estimates.

The LA_LINEAR_EQUATION function may also be used to solve for multiple systems of linear equations, with each column of *B* representing a different set of equations. In this case, the result is a *k*-by-*n* array where each of the *k* columns represents the solution vector for that set of equations.

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

Syntax

Result = LA_LINEAR_EQUATION(*Array*, *B* [, BACKWARD_ERROR=*variable*] [, /DOUBLE] [, FORWARD_ERROR=*variable*] [, STATUS=*variable*])

Return Value

The result is an *n*-element vector or *k*-by-*n* array.

Arguments

Array

The *n*-by-*n* array of the linear system *AX* = *B*.

B

An *n*-element input vector containing the right-hand side of the linear system, or a *k*-by-*n* array, where each of the *k* columns represents a different linear system.

Keywords

BACKWARD_ERROR

Set this keyword to a named variable that will contain the relative backward error estimate for each linear system. If *B* is a vector containing a single linear system, then BACKWARD_ERROR will be a scalar. If *B* is an array containing *k* linear systems, then BACKWARD_ERROR will be a *k*-element vector. The backward error is the smallest relative change in any element of *A* or *B* that makes *X* an exact solution.

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set DOUBLE = 0 to use single-precision for computations and to return a single-precision (real or complex) result. The default is /DOUBLE if Array is double precision, otherwise the default is DOUBLE = 0.

FORWARD_ERROR

Set this keyword to a named variable that will contain the estimated forward error bound for each linear system. If *B* is a vector containing a single linear system, then FORWARD_ERROR will be a scalar. If *B* is an array containing *k* linear systems, then FORWARD_ERROR will be a *k*-element vector. For each linear system, if *Xtrue* is the true solution corresponding to *X*, then the forward error is an estimated upper bound for the magnitude of the largest element in (*X* - *Xtrue*) divided by the magnitude of the largest element in *X*.

STATUS

Set this keyword to a named variable that will contain the status of the computation. Possible values are:

- STATUS = 0: The computation was successful.
- STATUS > 0: The computation failed because one of the diagonal elements of the LU decomposition is zero. The STATUS value specifies which value along the diagonal (starting at one) is zero.

Note

If STATUS is not specified, any error messages will be output to the screen.

Examples

Given the system of equations:

```
4u + 16000v + 17000w = 100.1
2u +     5v +     8w = 0.1
3u +     6v +    10w = 0.01
```

The following program can be used to find the solution:

```
PRO ExLA_LINEAR_EQUATION
; Define the coefficient array:
a = [[4, 16000, 17000], $
     [2, 5, 8], $
     [3, 6, 10]]

; Define the right-hand side vector b:
b = [100.1, 0.1, 0.01]

; Compute and print the solution to ax=b:
x = LA_LINEAR_EQUATION(a, b)
PRINT, 'LA_LINEAR_EQUATION solution:', X
end
```

When this program is compiled and run, IDL prints:

```
LA_LINEAR_EQUATION solution:
-0.397432    -0.334865     0.321148
```

The exact solution to 6 decimal places is [-0.397432, -0.334865, 0.321149].

Version History

Introduced 5.6

See Also

LA_LUDC, LA_LUMPROVE, LA_LUSOL

LA_LUDC

The LA_LUDC procedure computes the LU decomposition of an *n*-column by *m*-row array as:

$$A = P\,L\,U$$

where *P* is a permutation matrix, *L* is lower trapezoidal with unit diagonal elements (lower triangular if *n* = *m*), and *U* is upper trapezoidal (upper triangular if *n* = *m*).

LA_LUDC is based on the following LAPACK routines:

Output Type	LAPACK Routine
Float	sgetrf
Double	dgetrf
Complex	cgetrf
Double complex	zgetrf

Table 3-19: LAPACK Routine Basis for LA_LUDC

For details see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

LA_LUDC, *Array*, *Index* [, /DOUBLE] [, STATUS=*variable*]

Arguments

Array

A named variable containing the real or complex array to decompose. This procedure returns *Array* as its LU decomposition.

Index

An output vector with MIN(*m*, *n*) elements that records the row permutations which occurred as a result of partial pivoting. For 1 < *j* < MIN(*m*,*n*), row *j* of the matrix was interchanged with row *Index*[*j*].

Note

Row numbers within *Index* start at one rather than zero.

Keywords

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set DOUBLE = 0 to use single-precision for computations and to return a single-precision (real or complex) result. The default is /DOUBLE if *Array* is double precision, otherwise the default is DOUBLE = 0.

STATUS

Set this keyword to a named variable that will contain the status of the computation. Possible values are:

- STATUS = 0: The computation was successful.
- STATUS > 0: One of the diagonal elements of *U* is zero. The STATUS value specifies which value along the diagonal (starting at one) is zero.

Note

If STATUS is not specified, any error messages will output to the screen.

Examples

The following example uses the LU decomposition on a given array, then determines the residual error of using the resulting lower and upper arrays to recompute the original array:

```
PRO ExLA_LUDC
; Create a random array:
n = 20
seed = 12321
array = RANDOMN(seed, n, n)

; Compute LU decomposition.
aludc = array           ; make a copy
LA_LUDC, aludc, index
```

```
; Extract the lower and upper triangular arrays.
l = IDENTITY(n)
u = FLTARR(n, n)
FOR j = 1,n - 1 DO l[0:j-1,j] = aludc[0:j-1,j]
FOR j=0,n - 1 DO u[j:*,j] = aludc[j:*,j]

; Reconstruct array, but with rows permuted.
arecon = l ## u
; Adjust from LAPACK back to IDL indexing.
Index = Index - 1
; Permute the array rows back into correct order.
; Note that we need to loop in reverse order.
FOR i = n - 1,0,-1 DO BEGIN & $
     temp = arecon[*,i]
     arecon[*, i] = arecon[*,index[i]]
     arecon[*, index[i]] = temp
ENDFOR
PRINT, 'LA_LUDC Error:', MAX(ABS(arecon - array))
END
```

When this program is compiled and run, IDL prints:

```
LA_LUDC error: 4.76837e-007
```

Version History

Introduced 5.6

See Also

LA_LUMPROVE, LA_LUSOL, LUDC

LA_LUMPROVE

The LA_LUMPROVE function uses LU decomposition to improve the solution to a system of linear equations, *AX* = *B*, and provides optional error bounds and backward error estimates.

The LA_LUMPROVE function may also be used to improve the solutions for multiple systems of linear equations, with each column of *B* representing a different set of equations. In this case, the result is a *k*-by-*n* array where each of the *k* columns represents the improved solution vector for that set of equations.

LA_LUMPROVE is based on the following LAPACK routines:

Output Type	LAPACK Routine
Float	sgerfs
Double	dgetrfs
Complex	cgetrfs
Double complex	zgetrfs

Table 3-20: LAPACK Routine Basis for LA_LUMPROVE

For details see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

Result = LA_LUMPROVE(*Array*, *Aludc*, *Index*, *B*, *X*
[, BACKWARD_ERROR=*variable*] [, /DOUBLE]
[, FORWARD_ERROR=*variable*])

Return Value

The result is an *n*-element vector or *k*-by-*n* array.

Arguments

Array

The original *n*-by-*n* array of the linear system.

Aludc

The *n*-by-*n* LU decomposition of *Array*, created by the LA_LUDC procedure.

Index

An *n*-element input vector, created by the LA_LUDC procedure, containing the row permutations which occurred as a result of partial pivoting.

B

An *n*-element input vector containing the right-hand side of the linear system, or a *k*-by-*n* array, where each of the *k* columns represents a different linear system.

X

An *n*-element input vector, or a *k*-by-*n* array, containing the approximate solutions to the linear system, created by the LA_LUSOL function.

Keywords

BACKWARD_ERROR

Set this keyword to a named variable that will contain the relative backward error estimate for each linear system. If *B* is a vector containing a single linear system, then BACKWARD_ERROR will be a scalar. If *B* is an array containing *k* linear systems, then BACKWARD_ERROR will be a *k*-element vector. The backward error is the smallest relative change in any element of *A* or *B* that makes *X* an exact solution.

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set DOUBLE = 0 to use single-precision for computations and to return a single-precision (real or complex) result. The default is /DOUBLE if *Array* is double precision, otherwise the default is DOUBLE = 0.

FORWARD_ERROR

Set this keyword to a named variable that will contain the estimated forward error bound for each linear system. If *B* is a vector containing a single linear system, then FORWARD_ERROR will be a scalar. If *B* is an array containing *k* linear systems, then FORWARD_ERROR will be a *k*-element vector. For each linear system, if *Xtrue* is the true solution corresponding to *X*, then the forward error is an estimated upper bound for the magnitude of the largest element in (*X* - *Xtrue*) divided by the magnitude of the largest element in *X*.

Examples

The solution to a given system of equations can be derived and improved by using the following program:

```
PRO ExLA_LUMPROVE
; Define the coefficient array:
     a= [[4, 16000, 17000], $
     [2, 5, 8], $
     [3, 6, 10]]
; Compute the LU decomposition:
aludc = a
; make a copy
LA_LUDC, aludc, index

; Define the right-hand side vector B:
b = [100.1, 0.1, 0.01]
; Find the solution to Ax=b:
x = LA_LUSOL(aludc, index, b)
PRINT, 'LA_LUSOL Solution:', x

; Improve the solution:
xnew = LA_LUMPROVE(a, aludc, index, b, x)
PRINT, 'LA_LUMPROVE Solution:', xnew
END
```

When this program is compiled and run, IDL prints:

```
LA_LUSOL Solution:
-0.397355    -0.334742     0.321033
LA_LUMPROVE Solution:
-0.397432    -0.334865     0.321148
```

The exact solution to 6 decimal places is [-0.397432, -0.334865, 0.321149].

Version History

Introduced 5.6

See Also

LA_LUDC, LA_LUSOL, LUMPROVE

LA_LUSOL

The LA_LUSOL function is used in conjunction with the LA_LUDC procedure to solve a set of *n* linear equations in *n* unknowns, *AX* = *B*. The parameter *A* is not the original array, but its LU decomposition, created by the routine LA_LUDC.

The LA_LUSOL function may also be used to solve for multiple systems of linear equations, with each column of *B* representing a different set of equations. In this case, the result is a *k*-by-*n* array where each of the *k* columns represents the solution vector for that set of equations.

LA_LUSOL is based on the following LAPACK routines:

Output Type	LAPACK Routine
Float	sgetrs
Double	dgetrs
Complex	cgetrs
Double complex	zgetrs

Table 3-21: LAPACK Routine Basis for LA_LUSOL

For details see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

Result = LA_LUSOL(*A*, *Index*, *B* [, /DOUBLE])

Return Value

The result is an *n*-element vector or *k*-by-*n* array.

Arguments

A

The *n*-by-*n* LU decomposition of an array, created by the LA_LUDC procedure.

Note

LA_LUSOL cannot accept any non-square output generated by LA_LUDC.

Index

An *n*-element input vector, created by the LA_LUDC procedure, containing the row permutations which occurred as a result of partial pivoting.

B

An *n*-element input vector containing the right-hand side of the linear system, or a *k*-by-*n* array, where each of the *k* columns represents a different linear system.

Keywords

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set DOUBLE = 0 to use single-precision for computations and to return a single-precision (real or complex) result. The default is /DOUBLE if *A* is double precision, otherwise the default is DOUBLE = 0.

Examples

Given the system of equations:

```
4u + 16000v + 17000w = 100.1
2u +     5v +     8w = 0.1
3u +     6v +    10w = 0.01
```

find the solution can be derived by using the following program:

```
PRO ExLA_LUSOL
; Define the coefficient array:
a = [[4, 16000, 17000], $
     [2, 5, 8], $
     [3, 6, 10]]
; Compute the LU decomposition:
aludc = a
; make a copy
LA_LUDC, aludc, index

; Define the right-hand side vector B:
b = [100.1, 0.1, 0.01]

; Compute and print the solution to Ax=b:
x = LA_LUSOL(aludc, index, b)
PRINT, 'LA_LUSOL Solution:', x
END
```

When this program is compiled and run, IDL prints:

```
LA_LUSOL solution:    -0.397355    -0.334742     0.321033
```

The exact solution to 6 decimal places is [-0.397432, -0.334865, 0.321149].

Note

UNIX users may see slightly different output results.

Version History

Introduced 5.6

See Also

LA_LINEAR_EQUATION, LA_LUDC, LA_LUMPROVE, LUSOL

LA_SVD

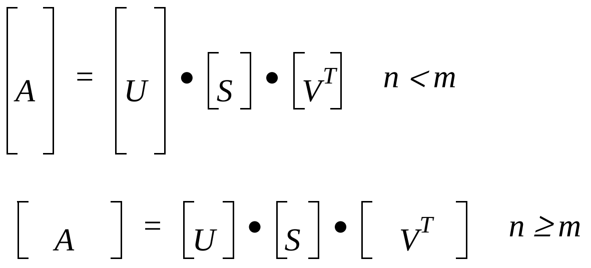
The LA_SVD procedure computes the singular value decomposition (SVD) of an *n*-columns by *m*-row array as the product of orthogonal and diagonal arrays:

A is real: $A = U\ S\ V^T$

A is complex: $A = U\ S\ V^H$

where *U* is an orthogonal array containing the left singular vectors, *S* is a diagonal array containing the singular values, and *V* is an orthogonal array containing the right singular vectors. The superscript *T* represents the transpose while the superscript *H* represents the Hermitian, or transpose complex conjugate.

If $n < m$ then *U* has dimensions (n x m), *S* has dimensions (n x n), and V^H has dimensions (n x n). If $n \geq m$ then *U* has dimensions (m x m), *S* has dimensions (m x m), and V^H has dimensions (n x m). The following diagram shows the array dimensions:



LA_SVD is based on the following LAPACK routines:

Output Type	LAPACK Routine	
	QR Iteration	Divide-and-conquer
Float	sgesvd	sgesdd
Double	dgesvd	dgesdd
Complex	cgesvd	cgesdd
Double complex	zgesvd	zgesdd

Table 3-22: LAPACK Routine Basis for LA_SVD

For details see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

LA_SVD, *Array, W, U, V* [, /DOUBLE] [, /DIVIDE_CONQUER]
[, STATUS=*variable*]

Arguments

Array

The real or complex array to decompose.

W

On output, *W* is a vector with MIN(*m*, *n*) elements containing the singular values.

U

On output, *U* is an orthogonal array with MIN(*m*, *n*) columns and *m* rows used in the decomposition of *Array*. If *Array* is complex then *U* will be complex, otherwise *U* will be real.

V

On output, *V* is an orthogonal array with MIN(*m*, *n*) columns and *n* rows used in the decomposition of *Array*. If *Array* is complex then *V* will be complex, otherwise *V* will be real.

Note

To reconstruct *Array*, you will need to take the transpose or Hermitian of *V*.

Keywords

DIVIDE_CONQUER

If this keyword is set, then the divide-and-conquer method is used to compute the singular vectors, otherwise, QR iteration is used. The divide-and-conquer method is faster at computing singular vectors of large matrices, but uses more memory and may produce less accurate singular values.

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set DOUBLE = 0 to use single-precision for computations and to return a single-precision (real or complex) result. The default is /DOUBLE if *Array* is double precision, otherwise the default is DOUBLE = 0.

STATUS

Set this keyword to a named variable that will contain the status of the computation. Possible values are:

- STATUS = 0: The computation was successful.
- STATUS > 0: The computation did not converge. The STATUS value specifies how many superdiagonals did not converge to zero.

Note

If STATUS is not specified, any error messages will output to the screen.

Examples

Construct a sample input array *A*, consisting of smoothed random values:

```
PRO ExLA_SVD
; Create a smoothed random array:
n = 100
m = 200
seed = 12321
a = SMOOTH(RANDOMN(seed, n, m, /DOUBLE), 5)

; Compute the SVD and check reconstruction error:
LA_SVD, a, w, u, v
arecon = u ## DIAG_MATRIX(w) ## TRANSPOSE(v)
PRINT, 'LA_SVD error:', MAX(ABS(arecon - a))

; Keep only the 15 largest singular values
wfiltered = w
wfiltered[15:*] = 0.0
; Reconstruct the array:
afiltered = u ## DIAG_MATRIX(wfiltered) ## TRANSPOSE(v)
percentVar = 100*(w^2)/TOTAL(w^2)
PRINT, 'LA_SVD Variance:', TOTAL(percentVar[0:14])
END
```

When this program is compiled and run, IDL prints:

```
LA_SVD error:  1.0103030e-014
LA_SVD variance:       82.802816
```

Note

More than 80% of the variance is contained in the 15 largest singular values.

Version History

Introduced 5.6

See Also

LA_CHOLDC, LA_LUDC, SVDC

LA_TRIDC

The LA_TRIDC procedure computes the LU decomposition of a tridiagonal (*n* x *n*) array as *Array* = *L U*, where *L* is a product of permutation and unit lower bidiagonal arrays, and *U* is upper triangular with nonzero elements only in the main diagonal and the first two superdiagonals.

LA_TRIDC is based on the following LAPACK routines:

Output Type	LAPACK Routine
Float	sgttrf
Double	dgttrf
Complex	cgttrf
Double complex	zgttrf

Table 3-23: LAPACK Routine Basis for LA_TRIDC

For more details, see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

LA_TRIDC, *AL*, *A*, *AU*, *U2*, *Index* [, /DOUBLE] [, STATUS=*variable*]

Arguments

AL

A named vector of length (*n* - 1) containing the subdiagonal elements of an array. This procedure returns *AL* as the (*n* - 1) elements of the lower bidiagonal array from the LU decomposition.

A

A named vector of length *n* containing the main diagonal elements of an array. This procedure returns *A* as the *n* diagonal elements of the upper array from the LU decomposition.

AU

A named vector of length (n - 1) containing the superdiagonal elements of an array. This procedure returns *AU* as the (n - 1) superdiagonal elements of the upper array.

U2

An output vector that contains the (n - 2) elements of the second superdiagonal of the upper array.

Index

An output vector that records the row permutations which occurred as a result of partial pivoting. For $1 < j < n$, row j of the matrix was interchanged with row *Index*[j].

Note

Row numbers within *Index* start at one rather than zero.

Keywords

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set DOUBLE = 0 to use single-precision for computations and to return a single-precision (real or complex) result. The default is /DOUBLE if *AL* is double precision, otherwise the default is DOUBLE = 0.

STATUS

Set this keyword to a named variable that will contain the status of the computation. Possible values are:

- STATUS = 0: The computation was successful.
- STATUS > 0: One of the diagonal elements of *U* is zero. The STATUS value specifies which value along the diagonal (starting at one) is zero.

Note

If STATUS is not specified, any error messages will output to the screen.

Examples

Create a test program to compute the LU decomposition of a tridiagonal array:

```
pro EX_LA_TRIDC
    ; Create a random tridiagonal array.
    n = 9
    seed = 12321
    AL = RANDOMN(seed, n-1)
    A  = RANDOMN(seed, n)
    AU = RANDOMN(seed, n-1)

    ; Construct tridiagonal array.
    Array = DIAG_MATRIX(AL, -1) + DIAG_MATRIX(A) + $
        DIAG_MATRIX(AU, 1)

    ; Compute the LU decomposition.
    LA_TRIDC, AL, A, AU, U2, Index

    ; Adjust from LAPACK back to IDL indexing.
    Index = Index - 1

    ; Create upper and lower arrays.
    Upper = DIAG_MATRIX(A) + $
        DIAG_MATRIX(AU, 1) + DIAG_MATRIX(U2, 2)
    Lower = DIAG_MATRIX(AL, -1) + IDENTITY(n)

    ; To conserve storage, LA_TRIDC keeps all lower diagonal
    ; elements in AL, regardless of row. The Index array
    ; tells which subdiagonals need to be shifted down.
    ; Loop starts at 1 since there aren't any subdiagonals
    ; to the left of the first diagonal element.
    for i = 1,n-2 do begin
        if (Index[i] ne i) then $
            Lower[0:i-1,[i,i+1]] = Lower[0:i-1,[i+1,i]]
    endfor

    ; Permute the row order.
    for i = n-2, 0, -1 do begin
        if (Index[i] ne i) then $
            Lower[*,[i,i+1]] = Lower[*,[i+1,i]]
    endfor

    ; Reconstruct the array and check the difference:
    Arecon = Lower ## Upper
    print, 'LA_TRIDC error:', MAX(ABS(Arecon - Array))
end
```

When this program is compiled and run, IDL prints:

```
LA_TRIDC error: 1.50427e-008
```

Version History

Introduced 5.6

See Also

LA_TRIMPROVE, LA_TRISOL

LA_TRIMPROVE

The LA_TRIMPROVE function improves the solution to a system of linear equations with a tridiagonal array, *AX* = *B*, and provides optional error bounds and backward error estimates.

The LA_TRIMPROVE function may also be used to improve the solutions for multiple systems of linear equations, with each column of *B* representing a different set of equations. In this case, the result is a *k*-by-*n* array where each of the *k* columns represents the improved solution vector for that set of equations.

LA_TRIMPROVE is based on the following LAPACK routines:

Output Type	LAPACK Routine
Float	sgtrfs
Double	dgtrfs
Complex	cgtrfs
Double complex	zgtrfs

Table 3-24: LAPACK Routine Basis for LA_TRIMPROVE

For details see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

Result = LA_TRIMPROVE(*AL*, *A*, *AU*, *DAL*, *DA*, *DAU*, *DU2*, *Index*, *B*, *X*
[, BACKWARD_ERROR=*variable*] [, /DOUBLE]
[, FORWARD_ERROR=*variable*])

Return Value

The result is an *n*-element vector or *k*-by-*n* array.

Arguments

AL

A vector of length (*n* - 1) containing the subdiagonal elements of the original array.

A

A vector of length *n* containing the main diagonal elements of the original array.

AU

A vector of length (*n* - 1) containing the superdiagonal elements of the original array.

DAL

The (*n* - 1) elements of the lower bidiagonal array, created by the LA_TRIDC procedure.

DA

The *n* diagonal elements of the upper triangular array, created by the LA_TRIDC procedure.

DAU

The (*n* - 1) superdiagonal elements of the upper triangular array, created by the LA_TRIDC procedure.

DU2

The (*n* - 2) elements of the second superdiagonal of the upper triangular array, created by the LA_TRIDC procedure.

Index

An input vector, created by the LA_TRIDC procedure, containing the row permutations which occurred as a result of partial pivoting.

B

An *n*-element input vector containing the right-hand side of the linear system, or a *k*-by-*n* array, where each of the *k* columns represents a different linear system.

X

An *n*-element input vector, or a *k*-by-*n* array, containing the approximate solutions to the linear system, created by the LA_TRISOL function.

Keywords

BACKWARD_ERROR

Set this keyword to a named variable that will contain the relative backward error estimate for each linear system. If *B* is a vector containing a single linear system, then BACKWARD_ERROR will be a scalar. If *B* is an array containing *k* linear systems, then BACKWARD_ERROR will be a *k*-element vector. The backward error is the smallest relative change in any element of *A* or *B* that makes *X* an exact solution.

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set DOUBLE = 0 to use single-precision for computations and to return a single-precision (real or complex) result. The default is /DOUBLE if *AL* is double precision, otherwise the default is DOUBLE = 0.

FORWARD_ERROR

Set this keyword to a named variable that will contain the estimated forward error bound for each linear system. If *B* is a vector containing a single linear system, then FORWARD_ERROR will be a scalar. If *B* is an array containing *k* linear systems, then FORWARD_ERROR will be a *k*-element vector. For each linear system, if *Xtrue* is the true solution corresponding to *X*, then the forward error is an estimated upper bound for the magnitude of the largest element in (*X* - *Xtrue*) divided by the magnitude of the largest element in *X*.

Examples

Given the tridiagonal system of equations:

```
-4t + u             =  6
 2t - 4u +  v       = -8
      2u - 4v + w   = -5
           2v -4w   =  8
```

the solution can be found and improved by using the following program:

```
PRO ExLA_TRIMPROVE
; Define array a:
aupper = [1, 1, 1]
adiag = [-4, -4, -4, -4]
alower = [2, 2, 2]
```

```
; Define right-hand side vector b:
b = [6, -8, -5, 8]

; Decompose a:
dlower = alower
darray = adiag
dupper = aupper
LA_TRIDC, dlower, darray, dupper, u2, index

; Compute and improve the solution:
x = LA_TRISOL(dlower, darray, dupper, u2, index, b)
xnew = LA_TRIMPROVE(Alower, Adiag, Aupper, $
     dlower, darray, dupper, u2, index, b, x)
PRINT, 'LA_TRISOL improved solution:'
PRINT, xnew
END
```

When this program is compiled and run, IDL prints:

```
LA_TRISOL improved solution:
-1.00000      2.00000      2.00000     -1.00000
```

Version History

Introduced 5.6

See Also

LA_TRIDC, LA_TRISOL

LA_TRIQL

The LA_TRIQL procedure uses the QL and QR variants of the implicitly-shifted QR algorithm to compute the eigenvalues and eigenvectors of a symmetric tridiagonal array. The LA_TRIRED routine can be used to reduce a real symmetric (or complex Hermitian) array to tridiagonal form suitable for input to this procedure.

LA_TRIQL is based on the following LAPACK routines:

Output Type	LAPACK Routine
Float	ssteqr
Double	dsteqr
Complex	csteqr
Double complex	zsteqr

Table 3-25: LAPACK Routine Basis for LA_TRIQL

For details see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

LA_TRIQL, D, E [, A] [, /DOUBLE] [, STATUS=*variable*]

Arguments

D

A named vector of length *n* containing the real diagonal elements, optionally created by the LA_TRIRED procedure. Upon output, *D* is replaced by a real vector of length *n* containing the eigenvalues.

E

The (*n* - 1) real subdiagonal elements, optionally created by the LA_TRIRED procedure. On output, the values within *E* are destroyed.

A

An optional named variable that returns the eigenvectors as a set of *n* row vectors. If the eigenvectors of a tridiagonal array are desired, *A* should be input as an identity array. If the eigenvectors of an array that has been reduced by LA_TRIRED are desired, *A* should be input as the Array output from LA_TRIRED. If *A* is not input, then eigenvectors are not computed. *A* may be either real or complex.

Keywords

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set DOUBLE = 0 to use single-precision for computations and to return a single-precision (real or complex) result. The default is DOUBLE = 0 if none of the inputs are double precision. If *A* is not input, then the default is /DOUBLE if *D* is double precision. If *A* is input, then the default is /DOUBLE if *A* is double precision (real or complex).

STATUS

Set this keyword to a named variable that will contain the status of the computation. Possible values are:

- STATUS = 0: The computation was successful.
- STATUS > 0: The algorithm failed to find all eigenvalues in 30*n* iterations. The STATUS value specifies how many elements of *E* have not converged to zero.

Note

If STATUS is not specified, any error messages will be output to the screen.

Examples

The following example program computes the eigenvalues and eigenvectors of a given symmetric array:

```
PRO ExLA_TRIQL
; Create a symmetric random array:
n = 4
seed = 12321
Array = RANDOMN(seed, n, n)
array = array + TRANSPOSE(array)
```

```
; Reduce to tridiagonal form
q = array    ; make a copy
LA_TRIRED, q, d, e

; Compute eigenvalues and eigenvectors
eigenvalues = d
eigenvectors = q
LA_TRIQL, eigenvalues, e, eigenvectors
PRINT, 'LA_TRIQL eigenvalues:'
PRINT, eigenvalues
END
```

When this program is compiled and run, IDL prints:

```
LA_TRIQL eigenvalues:
-2.87710    -0.663354      2.92018      3.59648
```

Version History

Introduced 5.6

See Also

LA_TRIRED, TRIQL

LA_TRIRED

The LA_TRIRED procedure reduces a real symmetric or complex Hermitian array to real tridiagonal form *T*. If the array is real symmetric then the decomposition is $A = Q\ T\ Q^T$, where *Q* is orthogonal. If the array is complex Hermitian then the decomposition is $A = Q\ T\ Q^H$, where *Q* is unitary. The superscript *T* represents the transpose while superscript *H* represents the Hermitian, or transpose complex conjugate.

LA_TRIRED is based on the following LAPACK routines:

Output Type	LAPACK Routine
Float	ssytrd, sorgtr
Double	dsytrd, dorgtr
Complex	chetrd, cungtr
Double complex	zhetrd, zungtr

Table 3-26: LAPACK Routine Basis for LA_TRIRED

For details see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

LA_TRIRED, *Array*, *D*, *E* [, /DOUBLE] [, /UPPER]

Arguments

Array

A named variable containing the real or complex array to decompose. Only the lower triangular portion of *Array* is used (or upper if the /UPPER keyword is set). This procedure returns *Array* as the real orthogonal (or complex unitary) *Q* array used to reduce the original array to tridiagonal form.

D

An *n*-element output vector containing the real diagonal elements of the tridiagonal array. Note that *D* is always real.

E

An (*n* - 1) element output vector containing the real subdiagonal elements of the tridiagonal array. Note that *E* is always real.

Keywords

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real) result. Set DOUBLE = 0 to use single-precision for computations and to return a single-precision (real) result. The default is /DOUBLE if *Array* is double precision, otherwise the default is DOUBLE = 0.

UPPER

If this keyword is set, then only the upper triangular portion of *Array* is used, and the upper triangular array is returned. The default is to use the lower triangular portion and return the lower triangular array.

Examples

See LA_TRIQL for an example of using this procedure.

Version History

Introduced 5.6

See Also

LA_TRIQL, TRIRED

LA_TRISOL

The LA_TRISOL function is used in conjunction with the LA_TRIDC procedure to solve a set of *n* linear equations in *n* unknowns, *AX* = *B*, where *A* is a tridiagonal array. The parameter *A* is input not as the original array, but as its LU decomposition, created by the routine LA_TRIDC.

The LA_TRISOL function may also be used to solve for multiple systems of linear equations, with each column of *B* representing a different set of equations. In this case, the result is a *k*-by-*n* array where each of the *k* columns represents the solution vector for that set of equations.

LA_TRISOL is based on the following LAPACK routines:

Output Type	LAPACK Routine
Float	sgttrs
Double	dgttrs
Complex	cgttrs
Double complex	zgttrs

Table 3-27: LAPACK Routine Basis for LA_TRISOL

For details see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

Result = LA_TRISOL(*AL*, *A*, *AU*, *U2*, *Index*, *B* [, /DOUBLE])

Return Value

The result is an *n*-element vector or *k*-by-*n* array.

Arguments

AL

The (*n* - 1) elements of the lower bidiagonal array, created by the LA_TRIDC procedure.

A

The *n* diagonal elements of the upper triangular array, created by the LA_TRIDC procedure.

AU

The (*n* - 1) superdiagonal elements of the upper triangular array, created by the LA_TRIDC procedure.

U2

The (*n* - 2) elements of the second superdiagonal of the upper triangular array, created by the LA_TRIDC procedure.

Index

An input vector, created by the LA_TRIDC procedure, containing the row permutations which occurred as a result of partial pivoting.

B

An *n*-element input vector containing the right-hand side of the linear system, or a *k*-by-*n* array, where each of the *k* columns represents a different linear system.

Keywords

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set DOUBLE = 0 to use single-precision for computations and to return a single-precision (real or complex) result. The default is /DOUBLE if *AL* is double precision, otherwise the default is DOUBLE = 0.

Example

For an example of using this routine see LA_TRIMPROVE.

Version History

Introduced 5.6

See Also

LA_TRIDC, LA_TRIMPROVE, TRISOL

MAP_PROJ_FORWARD

The MAP_PROJ_FORWARD function transforms map coordinates from longitude and latitude to Cartesian (x, y) coordinates, using either the !MAP system variable or a supplied map projection structure.

Syntax

Result = MAP_PROJ_FORWARD(*Longitude* [, *Latitude*]
[, CONNECTIVITY=*vector*] [, MAP_STRUCTURE=*value*]
[, POLYGONS=*variable*] [, POLYLINES=*variable*] [, /RADIANS])

Return Value

The result is a (2, *n*) array containing the Cartesian (x, y) coordinates.

Note

If the POLYGONS or POLYLINES keyword is present, the number of points in the result may be different than the number of input points, depending upon whether clipping and splitting occurs.

Arguments

Longitude

An *n*-element vector containing the longitude values. If the *Latitude* argument is omitted, *Longitude* must be a (2, *n*) array of longitude and latitude pairs.

Latitude

An *n*-element vector containing latitude values. If this argument is omitted, *Longitude* must be a (2, *n*) array of longitude and latitude pairs.

Keywords

CONNECTIVITY

Set this keyword to a vector containing an input connectivity list for polygons or polylines. The CONNECTIVITY keyword allows you to specify multiple polygons or polylines using a single array. The CONNECTIVITY list is a one-dimensional integer array of the form:

$$\left[m_1, i_0, i_1, \ldots, i_{m_1 - 1}, m_2, i_0, i_1, \ldots, i_{m_2 - 1}, \ldots, m_n, i_0, i_1, \ldots, i_{m_n - 1}\right]$$

where each m_j is an integer specifying the number of vertices that define the polyline or polygon (the *vertex count*), and each associated set of $i_0 \ldots i_{m-1}$ are indices into the arrays of vertices specified by the *Longitude* and *Latitude* arguments.

For example, to draw polylines between the first, third, and sixth longitude and latitude values and the fourth, sixth, ninth, and tenth longitude and latitude values, set the CONNECTIVITY array equal to `[3,0,2,5,4,3,5,8,9]`.

To ignore a set of entries in the CONNECTIVITY array, set the vertex count, m_j, equal to zero. (Note that if you set an *m* equal to zero, you must remove the associated set of $i_0 \ldots i_{m-1}$ values as well.) To ignore the remaining entries in the CONNECTIVITY array, set the vertex count, m_j, equal to -1.

This keyword is ignored if neither POLYGONS nor POLYLINES are present.

MAP_STRUCTURE

Set this keyword to a !MAP structure variable containing the projection parameters, as constructed by the MAP_PROJ_INIT. If this keyword is omitted, the !MAP system variable is used.

POLYGONS

Set this keyword to a named variable that will contain a connectivity array of the form described above in the CONNECTIVITY keyword.

If this keyword is present, the arrays specified by the *Longitude* and *Latitude* arguments are assumed to be the vertices of a closed polygon. In this case, polygon clipping and splitting is performed in addition to the map transform, and the connectivity array is returned in the specified variable. If this keyword is not present, the arrays specified by the *Longitude* and *Latitude* arguments are assumed to be independent points and no clipping or splitting is performed.

POLYLINES

Set this keyword to a named variable that will contain a connectivity array of the form described above in the CONNECTIVITY keyword.

If this keyword is present, the arrays specified by the *Longitude* and *Latitude* arguments are assumed to be the vertices of a polyline. In this case, polyline clipping and splitting is performed in addition to the map transform, and the connectivity array is returned in the specified variable.

If this keyword is not present, the arrays specified by the *Longitude* and *Latitude* arguments are assumed to be independent points and no clipping or splitting is performed.

RADIANS

Set this keyword to indicate that the input longitude and latitude coordinates are in radians. By default, coordinates are assumed to be in degrees.

Examples

The following example creates a latitude and longitude grid with labels for the Goodes Homolosine map projection.

```
; Helper function. Constructs the polyline objects.
PRO Ex_Map_AddPolyline, label, $
  gridLon, gridLat, sMap, oModel, oContainer, oFont, $
  LONGITUDE = longitude

longitude = KEYWORD_SET(longitude)

  ; Transform from lat/lon to X/Y cartesian.
gridUV = MAP_PROJ_FORWARD(gridLon, gridLat, $
    MAP=sMap, POLYLINES = gridPoly)
IF (N_ELEMENTS(gridUV) LT 2) THEN $
    RETURN

  ; Construct label object if desired.
IF (label NE '') THEN BEGIN
    oLabel = OBJ_NEW('IDLgrText', label, $
      ALIGN = longitude ? 0.5 : 1, $
      FONT = oFont, VERTICAL_ALIGN=0.5)
    oContainer->Add, oLabel
ENDIF
```

```
; Create the polyline object.
oModel->Add, OBJ_NEW('IDlgrPolyline', gridUV, $
    LABEL_OBJ = oLabel, $
    LABEL_OFFSET = longitude ? 0.35 : 0, $
    /USE_LABEL_ORIENTATION, /USE_TEXT_ALIGN, $
    POLYLINE = gridPoly)

END

; Main function. Creates a grid over a map projection.
PRO Ex_Map_Proj_Forward

; Construct !MAP structure containing the projection.
sMap = MAP_PROJ_INIT('Goodes Homolosine')

; Create a graphics model to hold the visualizations.
oModel = OBJ_NEW('IDLgrModel')
oContainer = OBJ_NEW('IDL_Container')
oFont = OBJ_NEW('IDLgrFont', SIZE = 4)
oContainer -> Add, oFont
deg = STRING(176b)  ; degrees symbol in Truetype

; Latitude lines.
gridLon = DINDGEN(361) - 180
latitude = 15*(INDGEN(11) - 5)

FOR i = 0,(N_ELEMENTS(latitude) - 1) DO BEGIN
  lat = latitude[i]
  gridLat = REPLICATE(lat, 361)

  ; Create the latitude label.
  label = (lat EQ 0) ? 'Equ' : $
    STRTRIM(ABS(lat),2) + deg + (['N','S'])[lat LT 0]
  Ex_Map_Addpolyline, label, gridLon, gridLat, $
    sMap, oModel, oContainer, oFont
ENDFOR

; Longitude lines.
gridLat = DINDGEN(181) - 90

; Add in some extra lines for the Goode projections.
longitude = [20*(DINDGEN(18) - 9), $
  -179.999d, -20.001d, -100.001d, -40.001d, 80.001d]

FOR i = 0,N_ELEMENTS(longitude) - 1 DO BEGIN
  lon = longitude[i]
  gridLon = REPLICATE(lon, 181)
```

```
      ; Create the longitude label.
      label = STRTRIM(ROUND(ABS(lon)),2) + deg
      IF ((lon mod 180) NE 0) THEN $
        label = label + (['E','W'])[lon LT 0]
      IF (lon NE FIX(lon)) THEN label = ''

      Ex_Map_Addpolyline, label, gridLon, gridLat, $
        sMap, oModel, oContainer, oFont, /LONGITUDE
    ENDFOR

    ; Visualize our map projection.
    XOBJVIEW, oModel, SCALE = 0.9, /BLOCK

    ; Clean up our objects.
    OBJ_DESTROY, [oModel, oContainer]

    END
```

Version History

Introduced: 5.6

See Also

MAP_PROJ_INIT, MAP_PROJ_INVERSE

MAP_PROJ_INIT

The MAP_PROJ_INIT function initializes a mapping projection, using either IDL's own map projections or map projections from the U.S. Geological Survey's General Cartographic Transformation Package (GCTP). GCTP version 2.0 is included with IDL.

Note

The !MAP system variable is unaffected by MAP_PROJ_INIT. To use the map projection returned by MAP_PROJ_INIT for direct or object graphics, use the MAP_PROJ_FORWARD and MAP_PROJ_INVERSE functions to convert longitude/latitude values into Cartesian (x, y) coordinates before visualization.

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

Syntax

Result = MAP_PROJ_INIT(*Projection* [, DATUM=*value*] [, /GCTP] [, LIMIT=*vector*] [, /RADIANS] [, /RELAXED])

Keywords—Projection Parameters:

[, CENTER_AZIMUTH=*value*] [, CENTER_LATITUDE=*value*]
[, CENTER_LONGITUDE=*value*] [, FALSE_EASTING=*value*]
[, FALSE_NORTHING=*value*] [, HEIGHT=*value*]
[, HOM_AZIM_LONGITUDE=*value*] [, HOM_AZIM_ANGLE=*value*]
[, HOM_LATITUDE1=*value*] [, HOM_LATITUDE2=*value*]
[, HOM_LONGITUDE1=*value*] [, HOM_LONGITUDE2=*value*]
[, IS_ZONES=*value*] [, IS_JUSTIFY=*value*] [, MERCATOR_SCALE=*value*]
[, OEA_ANGLE=*value*] [, OEA_SHAPEM=*value*] [, OEA_SHAPEN=*value*]
[, ROTATION=*value*] [, SEMIMAJOR_AXIS=*value*] [, SEMIMINORAXIS=*value*]
[, SOM_INCLINATION=*value*] [, SOM_LONGITUDE=*value*]
[, SOM_PERIOD=*value*] [, SOM_RATIO=*value*] [, SOM_FLAG=*value*]
[, SOM_LANDSAT_NUMBER=*value*] [, SOM_LANDSAT_PATH=*value*]
[, SPHERE_RADIUS=*value*] [, STANDARD_PARALLEL=*value*]
[, STANDARD_PAR1=*value*] [, STANDARD_PAR2=*value*] [, SAT_TILT=*value*]
[, TRUE_SCALE_LATITUDE=*value*] [, ZONE=*value*]

Return Value

The result is a !MAP structure containing the map parameters, which can be used as input to the map transformation functions MAP_PROJ_FORWARD and MAP_PROJ_INVERSE.

Arguments

Projection

Set this argument to either a projection index or a scalar string containing the name of the map projection, as described in following tables:

#	Projection Name	Allowed Keyword Parameters
1	Stereographic	SPHERE_RADIUS, CENTER_LONGITUDE, CENTER_LATITUDE, ROTATION
2	Orthographic	SPHERE_RADIUS, CENTER_LONGITUDE, CENTER_LATITUDE, ROTATION
3	Lambert Conic	SPHERE_RADIUS, STANDARD_PAR1, STANDARD_PAR2, CENTER_LONGITUDE, CENTER_LATITUDE
4	Lambert Azimuthal	SPHERE_RADIUS, CENTER_LONGITUDE, CENTER_LATITUDE, ROTATION
5	Gnomonic	SPHERE_RADIUS, CENTER_LONGITUDE, CENTER_LATITUDE, ROTATION
6	Azimuthal Equidistant	SPHERE_RADIUS, CENTER_LONGITUDE, CENTER_LATITUDE, ROTATION

Table 3-28: IDL Projections

#	Projection Name	Allowed Keyword Parameters
7	Satellite	SPHERE_RADIUS, HEIGHT, SAT_TILT, CENTER_LONGITUDE, CENTER_LATITUDE, ROTATION
8	Cylindrical	SPHERE_RADIUS, CENTER_AZIMUTH, CENTER_LONGITUDE, CENTER_LATITUDE, ROTATION
9	Mercator	SPHERE_RADIUS, CENTER_AZIMUTH, CENTER_LONGITUDE, CENTER_LATITUDE, ROTATION
10	Mollweide	SPHERE_RADIUS, CENTER_AZIMUTH, CENTER_LONGITUDE, CENTER_LATITUDE, ROTATION
11	Sinusoidal	SPHERE_RADIUS, CENTER_AZIMUTH, CENTER_LONGITUDE, CENTER_LATITUDE, ROTATION
12	Aitoff	SPHERE_RADIUS, CENTER_LONGITUDE, CENTER_LATITUDE, ROTATION
13	Hammer Aitoff	SPHERE_RADIUS, CENTER_LONGITUDE, CENTER_LATITUDE, ROTATION
14	Albers Equal Area Conic	SPHERE_RADIUS, STANDARD_PAR1, STANDARD_PAR2, CENTER_LONGITUDE, CENTER_LATITUDE
15	Transverse Mercator	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, MERCATOR_SCALE, CENTER_LONGITUDE, CENTER_LATITUDE, ROTATION
16	Miller Cylindrical	SPHERE_RADIUS, CENTER_AZIMUTH, CENTER_LONGITUDE, CENTER_LATITUDE, ROTATION

Table 3-28: IDL Projections (Continued)

#	Projection Name	Allowed Keyword Parameters
17	Robinson	SPHERE_RADIUS, CENTER_AZIMUTH, CENTER_LONGITUDE, CENTER_LATITUDE, ROTATION
18	Lambert Ellipsoid Conic	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, STANDARD_PAR1, STANDARD_PAR2, CENTER_LONGITUDE, CENTER_LATITUDE
19	Goodes Homolosine	SPHERE_RADIUS, CENTER_LONGITUDE

Table 3-28: IDL Projections (Continued)

The following are GCTP projections:

#	Projection Name	Allowed Keyword Parameters
101	UTM	CENTER_LONGITUDE, CENTER_LATITUDE, ZONE
102	State Plane	ZONE
103	Albers Equal Area	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, STANDARD_PAR1, STANDARD_PAR2, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
104	Lambert Conformal Conic	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, STANDARD_PAR1, STANDARD_PAR2, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
105	Mercator	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, CENTER_LONGITUDE, TRUE_SCALE_LATITUDE, FALSE_EASTING, FALSE_NORTHING

Table 3-29: GCTP Projections

#	Projection Name	Allowed Keyword Parameters
106	Polar Stereographic	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
107	Polyconic	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
108	Equidistant Conic A	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, STANDARD_PARALLEL, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
208	Equidistant Conic B	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, STANDARD_PAR1, STANDARD_PAR2, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
109	Transverse Mercator	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, MERCATOR_SCALE, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
110	Stereographic	SPHERE_RADIUS, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
111	Lambert Azimuthal	SPHERE_RADIUS, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING

Table 3-29: GCTP Projections (Continued)

#	Projection Name	Allowed Keyword Parameters
112	Azimuthal	SPHERE_RADIUS, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
113	Gnomonic	SPHERE_RADIUS, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
114	Orthographic	SPHERE_RADIUS, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
115	Near Side Perspective	SPHERE_RADIUS, HEIGHT, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
116	Sinusoidal	SPHERE_RADIUS, CENTER_LONGITUDE, FALSE_EASTING, FALSE_NORTHING
117	Equirectangular	SPHERE_RADIUS, CENTER_LONGITUDE, TRUE_SCALE_LATITUDE, FALSE_EASTING, FALSE_NORTHING
118	Miller Cylindrical	SPHERE_RADIUS, CENTER_LONGITUDE, FALSE_EASTING, FALSE_NORTHING
119	Van der Grinten	SPHERE_RADIUS, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING

Table 3-29: GCTP Projections (Continued)

#	Projection Name	Allowed Keyword Parameters
120	Hotine Oblique Mercator A	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, MERCATOR_SCALE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING, HOM_LONGITUDE1, HOM_LATITUDE1, HOM_LONGITUDE2, HOM_LATITUDE2
220	Hotine Oblique Mercator B	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, MERCATOR_SCALE, HOM_AZIM_ANGLE, HOM_AZIM_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
121	Robinson	SPHERE_RADIUS, CENTER_LONGITUDE, FALSE_EASTING, FALSE_NORTHING
122	Space Oblique Mercator A	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, SOM_INCLINATION, SOM_LONGITUDE, FALSE_EASTING, FALSE_NORTHING, SOM_PERIOD, SOM_RATIO, SOM_FLAG
222	Space Oblique Mercator B	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, SOM_LANDSAT_NUMBER, SOM_LANDSAT_PATH, FALSE_EASTING, FALSE_NORTHING
123	Alaska Conformal	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, FALSE_EASTING, FALSE_NORTHING
124	Interrupted Goode	SPHERE_RADIUS
125	Mollweide	SPHERE_RADIUS, CENTER_LONGITUDE, FALSE_EASTING, FALSE_NORTHING
126	Interrupted Mollweide	SPHERE_RADIUS

Table 3-29: GCTP Projections (Continued)

#	Projection Name	Allowed Keyword Parameters
127	Hammer	SPHERE_RADIUS, CENTER_LONGITUDE, FALSE_EASTING, FALSE_NORTHING
128	Wagner IV	SPHERE_RADIUS, CENTER_LONGITUDE, FALSE_EASTING, FALSE_NORTHING
129	Wagner VII	SPHERE_RADIUS, CENTER_LONGITUDE, FALSE_EASTING, FALSE_NORTHING
130	Oblated Equal Area	SPHERE_RADIUS, OEA_SHAPEM, OEA_SHAPEN, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING, OEA_ANGLE
131	Integerized Sinusoidal	SPHERE_RADIUS, CENTER_LONGITUDE, FALSE_EASTING, FALSE_NORTHING, IS_ZONES, IS_JUSTIFY

Table 3-29: GCTP Projections (Continued)

Keywords

Note

The following keywords apply to all projections.

DATUM

Set this keyword to either an integer code or a scalar string containing the name of the datum to use for the ellipsoid. The default value depends upon the projection selected, but is either the Clarke 1866 ellipsoid (datum 0), or a sphere of radius 6370.997 km (datum 19).

The following datums (or spheroids) are available for use with the DATUM keyword:

Index	Name	Semimajor axis (m)	Semiminor axis (m)
0	Clarke 1866	6378206.4	6356583.8
1	Clarke 1880	6378249.145	6356514.86955
2	Bessel	6377397.155	6356078.96284
3	International 1967	6378157.5	6356772.2
4	International 1909	6378388.0	6356911.94613
5	WGS 72	6378135.0	6356750.519915
6	Everest	6377276.3452	6356075.4133
7	WGS 66	6378145.0	6356759.769356
8	GRS 1980/WGS 84	6378137.0	6356752.31414
9	Airy	6377563.396	6356256.91
10	Modified Everest	6377304.063	6356103.039
11	Modified Airy	6377340.189	6356034.448
12	Walbeck	6378137.0	6356752.314245
13	Southeast Asia	6378155.0	6356773.3205
14	Australian National	6378160.0	6356774.719
15	Krassovsky	6378245.0	6356863.0188
16	Hough	6378270.0	6356794.343479
17	Mercury 1960	6378166.0	6356784.283666
18	Modified Mercury 1968	6378150.0	6356768.337303
19	Sphere	6370997.0	6370997.0

Table 3-30: Datums available for use by MAP_PROJ_INIT.

Note

For many projections, you can specify your own datum by using either the SEMIMAJOR_AXIS and SEMIMINOR_AXIS or the SPHERE_RADIUS keywords.

GCTP

Set this keyword to indicate that the GCTP library should be used for the projection. By default, MAP_PROJ_INIT uses the IDL projection library. This keyword is ignored if the projection exists only in one system (GCTP or IDL), or if the *Projection* argument is specified as an index.

LIMIT

Set this keyword to a four-element vector of the form

```
[Latmin, Lonmin, Latmax, Lonmax]
```

that specifies the boundaries of the region to be mapped. (*Lonmin*, *Latmin*) and (*Lonmax*, *Latmax*) are the longitudes and latitudes of two points diagonal from each other on the region's boundary.

Note

When using MAP_PROJ_FORWARD, if the longitude range in LIMIT is less than or equal to 180 degrees, map clipping is performed in lat/lon coordinates *before* the transform. If the longitude range is greater than 180 degrees, map clipping is done in Cartesian coordinates *after* the transform. For non-cylindrical projections, clipping *after* the transformation to Cartesian coordinates means that some lat/lon points that fall outside the bounds specified by LIMIT may not be clipped. This occurs when the *transformed* lat/lon points fall inside the cartesian clipping rectangle.

RADIANS

Set this keyword to indicate that all parameters that represent angles are specified in radians rather than degrees.

RELAXED

If this keyword is set, any projection parameters which do not apply to the specified projection will be quietly ignored. By default, MAP_PROJ_INIT will issue errors for parameters that do not apply to the specified projection.

Projection Keywords

The following keywords apply only to some projections. Consult the list under "Projection" on page 397 to determine which keywords apply to the projection you have selected.

CENTER_AZIMUTH

Set this keyword to the angle of the central azimuth, in degrees east of North. The default is 0 degrees. The pole is placed at an azimuth of CENTRAL_AZIMUTH degrees counterclockwise of North, as specified by the ROTATION keyword.

CENTER_LATITUDE

Set this keyword to the latitude of the point on the earth's surface to be mapped to the center of the projection plane. Latitude is measured in degrees North of the equator and must be in the range: -90 to +90. The default value is zero.

CENTER_LONGITUDE

Set this keyword to the longitude of the point on the earth's surface to be mapped to the center of the map projection. Longitude is measured in degrees east of the Greenwich meridian and must be in the range: -360 to +360. The default value is zero.

FALSE_EASTING

Set this keyword to the false easting value (in meters) to be added to each x coordinate for the forward transform, or subtracted from each x coordinate for the inverse transform.

FALSE_NORTHING

Set this keyword to the false northing value (in meters) to be added to each y coordinate for the forward transform, or subtracted from each y coordinate for the inverse transform.

HEIGHT

Set this keyword to the height (in meters) above the earth's surface for satellite projections.

HOM_AZIM_LONGITUDE

Set this keyword to the longitude of the central meridian point where the azimuth occurs.

HOM_AZIM_ANGLE

Set this keyword to the azimuth angle, measured in degrees or radians, east of a north-south line that intersects the center line. The center line is defined as the great circle path along which the Mercator cylinder touches the sphere.

HOM_LATITUDE1

Set this keyword to the latitude of the first point on the center line. The center line is defined as the great circle path along which the Mercator cylinder touches the sphere.

HOM_LATITUDE2

Set this keyword to the latitude of the second point on the center line. The center line is defined as the great circle path along which the Mercator cylinder touches the sphere.

HOM_LONGITUDE1

Set this keyword to the longitude of the first point on the center line. The center line is defined as the great circle path along which the Mercator cylinder touches the sphere.

HOM_LONGITUDE2

Set this keyword to the longitude of the second point on the center line. The center line is defined as the great circle path along which the Mercator cylinder touches the sphere.

IS_ZONES

Set this keyword to the number of longitudinal zones to include in the projection.

IS_JUSTIFY

Set this keyword to a flag indicating what to do with rows with an odd number of columns. The possible values are:

Value	Description
0	Indicates the extra column is on the right of the projection Y axis.
1	Indicates the extra column is on the left of the projection Y axis.
2	Calculate an even number of columns.

Table 3-31: IS_JUSTIFY Keyword Values

MERCATOR_SCALE

Set this keyword to the scale factor at the central meridian (Transverse Mercator projection) or the center of the projection (Hotine Oblique Mercator projection). For the Transverse Mercator projection, the default scale is 0.9996.

OEA_ANGLE

Set this keyword to the Oblated Equal Area oval rotation angle.

OEA_SHAPEM

Set this keyword to the Oblated Equal Area shape parameter *m*. The value of OEA_SHAPEM determines the horizontal flatness of the oblong region, and is usually set to a value between one and three.

OEA_SHAPEN

Set this keyword to the Oblated Equal Area oval shape parameter *n*. The value of OEA_SHAPEN determines the vertical flatness of the oblong region, and is usually set to a value between one and three.

Note

Setting both OEA_SHAPEM and OEA_SHAPEN equal to two is equivalent to using the Lambert Azimuthal projection.

ROTATION

Set this keyword to the angle through which the North direction should be rotated around the line between the earth's center and the point (CENTER_LONGITUDE, CENTER_LATITUDE). ROTATION is measured in degrees with the positive direction being clockwise rotation around the line. Values should be in the range -180 to +180. The default value is zero.

Note

If the center of the map is at the North pole, North is in the direction CENTER_LONGITUDE + 180. If the origin is at the South pole, North is in the direction CENTER_LONGITUDE.

SEMIMAJOR_AXIS

Set this keyword to the length (in meters) of the semimajor axis of the reference ellipsoid. The default is either the Clarke 1866 datum (6378206.4 m) or the Sphere radius (6370997 m), depending upon the projection.

SEMIMINOR_AXIS

Set this keyword to the length (in meters) of the semiminor axis of the reference ellipsoid. The default is either the Clarke 1866 datum (6356583.8 m) or the Sphere radius (6370997 m), depending upon the projection.

SOM_INCLINATION

Set this keyword to the orbit inclination angle of the ascending node, counter-clockwise from equator.

SOM_LONGITUDE

Set this keyword to the longitude of the ascending orbit at the equator.

SOM_PERIOD

Set this keyword to the period in minutes of the satellite revolution.

SOM_RATIO

Set this keyword to the Landsat ratio to compensate for confusion at the northern end of orbit. A typical value is 0.5201613.

SOM_FLAG

Set this keyword to the end of path flag for Landsat, where 0 is the start and 1 is the end.

SOM_LANDSAT_NUMBER

Set this keyword to the Landsat satellite number.

SOM_LANDSAT_PATH

Set this keyword to the Landsat path number (use 1 for Landsat 1, 2 and 3; use 2 for Landsat 4, 5 and 6).

SPHERE_RADIUS

Set this keyword to the radius (in meters) of the reference sphere. The default is 6370997 m.

STANDARD_PARALLEL

Set this keyword to the latitude of the standard parallel along which the scale is true.

STANDARD_PAR1

Set this keyword to the latitude of the first standard parallel along which the scale is true.

STANDARD_PAR2

Set this keyword to the latitude of the second standard parallel along which the scale is true.

SAT_TILT

Set this keyword to the downward tilt in degrees of the camera, in degrees from the projection horizontal.

TRUE_SCALE_LATITUDE

Set this keyword to the latitude of true scale.

ZONE

Set this keyword to an integer giving the zone for the GCTP UTM projection or GCTP State Plane projection.

Note

For the UTM projection, you may also use the CENTER_LONGITUDE and CENTER_LATITUDE keywords to set the zone. Internally, the ZONE value will be computed from the longitude and latitude.

Examples

See MAP_PROJ_FORWARD for an example of using this function.

Version History

Introduced: 5.6

See Also

MAP_PROJ_FORWARD, MAP_PROJ_INVERSE, MAP_SET

MAP_PROJ_INVERSE

The MAP_PROJ_INVERSE function transforms map coordinates from Cartesian (x, y) coordinates to longitude and latitude, using either the !MAP system variable or a supplied map projection variable.

Syntax

Result = MAP_PROJ_INVERSE (*X* [, *Y*] [, MAP_STRUCTURE=*value*] [, /RADIANS])

Return Value

The result is a (2, *n*) array containing the longitude/latitude coordinates.

Arguments

X

An *n*-element vector containing the x values. If the *Y* argument is omitted, *X* must be a (2, *n*) array of X and Y pairs.

Y

An *n*-element vector containing y values. If this argument is omitted, *X* must be a (2, *n*) array of X and Y pairs.

Keywords

MAP_STRUCTURE

Set this keyword to a !MAP structure variable containing the projection parameters, as constructed by the MAP_PROJ_INIT. If this keyword is omitted, the !MAP system variable is used.

RADIANS

Set this keyword to indicate that the returned longitude and latitude coordinates should be expressed in radians. By default, returned coordinates are expressed in degrees.

Version History

Introduced: 5.6

See Also

MAP_PROJ_FORWARD, MAP_PROJ_INIT

MATRIX_POWER

The MATRIX_POWER function computes the product of a matrix with itself. For example, the fifth power of array *A* is *A* # *A* # *A* # *A* # *A*. Negative powers are computed using the matrix inverse of the positive power.

Syntax

Result = MATRIX_POWER(*Array*, *N* [, /DOUBLE] [, STATUS=*value*])

Return Value

The result is a square array containing the value of the matrix raised to the specified power. A power of zero returns the identity matrix.

Arguments

Array

A square, two-dimensional array of any numeric type.

N

An integer representing the power. *N* may be positive or negative.

Keywords

DOUBLE

Set this keyword to return a double-precision result. Explicitly set this keyword equal to zero to return a single-precision result. The default return type depends upon the precision of *Array*.

Note

Computations are always performed using double-precision arithmetic.

STATUS

Set this keyword equal to a named variable that will contain the status of the matrix inverse for negative powers. Possible values are:

Value	Description
0	Successful completion.
1	Singular array (which indicates that the inversion is invalid).
2	Warning that a small pivot element was used and that significant accuracy was probably lost.

Table 3-32: STATUS Keyword Values

For non-negative powers, STATUS is always set to 0.

Example

Print an array to the one millionth power:

```
array = [ [0.401d, 0.600d], $
          [0.525d, 0.475d] ]
PRINT, MATRIX_POWER(array, 1e6)
```

IDL prints:

```
  2.4487434e+202  2.7960773e+202
  2.4465677e+202  2.7935929e+202
```

Version History

Introduced: 5.6

See Also

MATRIX_MULTIPLY, “Multiplying Arrays” in Chapter 22 of the *Using IDL* manual

PRODUCT

The PRODUCT function returns the product of elements within an array. The product of the array elements over a given dimension is returned if the *Dimension* argument is present. Because the product can easily overflow, the product is computed using double-precision arithmetic and the *Result* is double precision.

Tip

If your array has a mix of very large and very small values, the product may underflow or overflow during the computation, even though the final result would be within double-precision limits. In this case, you should not use PRODUCT, but instead compute the product by taking the logarithm, using the TOTAL function, and then taking the exponential: *Result* = EXP(TOTAL(ALOG(*Array*))).

Syntax

Result = PRODUCT(*Array* [, *Dimension*] [, /CUMULATIVE] [, /NAN])

Return Value

Returns the product of the elements of *Array*.

Arguments

Array

The array for which to compute the product. This array can be of any basic type except string.

Dimension

An optional argument specifying the dimension over which to compute the product, starting at one. If this argument is not present or zero, the product of all the array elements is returned. If this argument is present, the result is an array with one less dimension than *Array*. For example, if the dimensions of *Array* are N1, N2, N3, and *Dimension* is 2, the dimensions of the result are (N1, N3), and element (i,j) of the result contains the product:

$$R_{i,j} = \prod_{k=0}^{N_2-1} A_{i,k,j}$$

Keywords

CUMULATIVE

If this keyword is set, the result is an array of the same size as the input, with each element, *i*, containing the product of the input array elements 0 to *i*. This keyword also works with the *Dimension* parameter, in which case the cumulative product is performed over the given dimension.

NAN

Set this keyword to cause the routine to check for occurrences of the IEEE floating-point value NaN in the input data. Elements with the value NaN are treated as missing data with the value 1.

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 I, “Thread Pool Keywords” in the *IDL Reference Guide*.

Examples

To find the product of all elements in a one-dimensional array:

```
; Define a one-dimensional array:
array = [20, 10, 5, 5, 3]

; Find the product of the array elements:
prod = PRODUCT(array)

; Print the results:
PRINT, 'Product of Array = ', prod
```

IDL prints:

```
Product of Array =        15000.000
```

Now find the product of elements in a two-dimensional array:

```
; Define a two-dimensional array:
array = FINDGEN(4,4) + 1
```

```
; Find the product of all array elements:
prodAll = PRODUCT(array)

; Find the product along the first dimension:
prod1 = PRODUCT(array, 1)

; Find the product along the second dimension:
prod2 = PRODUCT(array, 2)

; Print the results:
PRINT, 'Product of all elements = ', prodAll
PRINT, 'Product along first dimension: '
PRINT, prod1
PRINT, 'Product along second dimension: '
PRINT, prod2
```

IDL prints:

```
Product of all elements   2.0922790e+013
Product along first dimension:
 24.000000       1680.0000       11880.000       43680.000
Product along second dimension:
 585.00000       1680.0000       3465.0000       6144.0000
```

Version History

Introduced: 5.6

See Also

FACTORIAL, TOTAL

REGISTER_CURSOR

The REGISTER_CURSOR procedure associates the given name with the given cursor information. This name can then be used with the IDLgrWindow::SetCurrentCursor method.

Syntax

REGISTER_CURSOR, *Name, Image*[, MASK=*value*] [, HOTSPOT=*value*] [, /OVERWRITE]

Arguments

Name

This argument sets the name to associate with this cursor. The name is case-insensitive. Once registered, the name can be used with the IDLgrWindow::SetCurrentCursor method.

Image

Set this argument to a 16 line by 16 column bitmap, contained in a 16-element short integer vector, specifying the cursor pattern. The offset from the upper-left pixel to the point that is considered the "hot spot" can be provided using the HOTSPOT keyword.

Keywords

MASK

This keyword can be used to simultaneously specify the mask that should be used. In the mask, bits that are set indicate bits in the IMAGE that should be seen and bits that are not are "masked out".

HOTSPOT

Set this keyword to a two-element vector specifying the [*x*, *y*] pixel offset of the cursor "hot spot", the point which is considered to be the mouse position, from the lower-left corner of the cursor image. The cursor image is displayed top-down (the first row is displayed at the top).

OVERWRITE

By default, if the cursor already exists, the values are not changed. By setting this keyword to true, the current cursor value is updated with the values provided by this routine call.

Version History

Introduced: 5.6

See Also

IDLgrWindow::SetCurrentCursor

SHMDEBUG

The SHMDEBUG function enables a debugging mode in which IDL prints an informational message (including a traceback) every time a variable created with the SHMVAR function loses its reference to the underlying memory segment created by SHMMAP. There are many reasons why a such a variable might lose its reference; some reasons have to do with the internal implementation of the IDL interpreter and are not obvious or visible to the IDL user.

Note

The SHMDEBUG debugging mode should be used for problem solving only, and should not be part of production code.

Syntax

Result = SHMDEBUG(*Enable*)

Return Value

SHMDEBUG returns the previous setting of the debugging state.

Arguments

Enable

Set this argument equal to a non-zero value to enable debugging, or to zero to disable debugging.

Examples

Create a memory segment, tie a variable to it, enable debugging, and then cause the variable to lose the reference:

```
old_debug = SHMDEBUG(1) ; Enable debug mode
SHMMAP, 'A', 100 ; 100 element floating vector
z = SHMVAR('A') ; Variable tied to segment
z[0] = FINDGEN(100) ; Does not lose reference
z = FINDGEN(100) ; Loses reference
% Variable released shared memory segment: A
% Released at: $MAIN$
```

The assignment `z[0] = FINDGEN(100)` explicitly uses subscripting to assign the FINDGEN value to the array. Under normal circumstances, using subscripting in this way on the left hand side of an assignment is inefficient and not recommended. In this case, however, it has the desirable side effect of causing the variable Z to maintain its connection to its existing underlying memory. In contrast, the second (normally more desirable) assignment without the subscript causes IDL to allocate different memory for the variable Z, with the side effect of losing the connection to the shared memory segment.

Version History

Introduced: 5.6

See Also

SHMMAP, SHMUNMAP, SHMVAR

SHMMAP

The SHMMAP procedure maps anonymous shared memory, or local disk files, into the memory address space of the currently executing IDL process. Mapped memory segments are associated with an IDL array specified by the user as part of the call to SHMMAP. The type and dimensions of the specified array determine the length of the memory segment.

The array can be of any type except pointer, object reference, or string. (Structure types are allowed as long as they do not contain any pointers, object references, or strings.) By default, the array type is single-precision floating-point; other types can be chosen by specifying the appropriate keyword.

Once such a memory segment exists, it can be tied to an actual IDL variable using the SHMVAR function, or unmapped using SHMUNMAP.

Why Use Mapped Memory?

- Shared memory is often used for interprocess communication. Any process that has a shared memory segment mapped into its address space is able to "see" any changes made by any other process that has access to the same segment. Shared memory is the default for SHMMAP, unless the FILENAME keyword is specified.
- Memory-mapped files allow you to treat the contents of a local disk file as if it were simple memory. Reads and writes to such memory are automatically written to the file by the operating system using its standard virtual memory mechanisms. Access to mapped files has the potential to be faster than standard Input/Output using Read/Write system calls because it does not go through the expensive system call interface, and because it does not require the operating system to copy data between user and kernel memory buffers when performing the I/O. However, it is not as general or flexible as the standard I/O mechanisms, and is therefore not a replacement for them.

Warning

Unlike most IDL functionality, incorrect use of SHMMAP can corrupt or even crash your IDL process. Proper use of these low level operating system features requires systems programming experience, and is not recommended for those without such experience. You should be familiar with the memory and file mapping features of your operating system and the terminology used to describe such features.

SHMMAP uses the facilities of the underlying operating system. Any of several alternatives may be used, as described in "Types Of Memory Segments" on page 430. SHMMAP uses the following rules, in the specified order, to determine which method to use:

1. If the FILENAME keyword is present, SHMMAP creates a memory mapped file segment.
2. If the SYSV keyword is used under UNIX, a System V shared memory segment is created or attached. Use of the SYSV keyword under Windows will cause an error to be issued.
3. If the LOCAL_MEMORY keyword is present, a local memory segment is created.
4. If none of the above options are specified, SHMMAP creates an anonymous shared memory segment. Under UNIX, this is done with Posix shared memory. Under Windows, the `CreateFileMapping()` system call is used.

Syntax

SHMMAP [, *SegmentName*] [, D_1, ..., D_8] [, /BYTE] [, /COMPLEX] [, /DCOMPLEX] [, /DESTROY_SEGMENT] [, DIMENSION=*value*] [, /DOUBLE] [, FILENAME=*value*] [, /FLOAT] [, GET_NAME=*value*] [, GET_OS_HANDLE=*value*] [, /INTEGER] [, /L64] [, /LONG] [, OFFSET=*value*] [, OS_HANDLE=*value*] [, /PRIVATE] [, SIZE=*value*] [, /SYSV] [, TEMPLATE=*value*] [, TYPE=*value*] [, /UINT] [, /UL64] [, /ULONG]

Arguments

SegmentName

A scalar string supplying the name by which IDL will refer to the shared memory segment. This name is only used by IDL, and does not necessarily correspond to the name used for the shared memory segment by the underlying operating system. See the discussion of the OS_HANDLE keyword for more information on the underlying operating system name. If *SegmentName* is not specified, IDL will generate a unique name. The *SegmentName* can be obtained using the GET_NAME keyword.

Di

The dimensions of the result. The D_i arguments can be either a single array containing the dimensions or a sequence of scalar dimensions. Up to eight dimensions can be specified.

Keywords

BYTE

Set this keyword to specify that the memory segment should be treated as a byte array.

COMPLEX

Set this keyword to specify that the memory segment should be treated as a complex, single-precision floating-point array.

DCOMPLEX

Set this keyword to specify that the memory segment should be treated as a complex, double-precision floating-point array.

DESTROY_SEGMENT

The UNIX anonymous shared memory mechanisms (Posix `shm_open()` and System V `shmget()`) create shared memory segments that are not removed from the operating system kernel until explicitly destroyed (or the system is rebooted). At any time, a client program can attach to such an existing segment, read or write to it, and then detach. This can be convenient in situations where the need for the shared memory is long lived, and programs that need it come and go. It also can create a problem, however, in that shared memory segments that are not explicitly destroyed can cause memory leaks in the operating system. Hence, it is important to properly destroy such segments when they are no longer required.

For UNIX anonymous shared memory (Posix or System V), the default behavior is for IDL to destroy any shared memory segments it created when the segments are unmapped, and not to destroy segments it did not create. The DESTROY_SEGMENT keyword is used to override this default: set DESTROY_SEGMENT to 1 (one) to indicate that IDL should destroy the segment when it is unmapped, or 0 (zero) to indicate that it should not destroy it. All such destruction occurs when the segment is unmapped (via the SHMUNMAP procedure) and not during the call to SHMMAP.

The DESTROY_SEGMENT keyword is ignored under the Windows operating system. Under UNIX, it is ignored for mapped files.

DIMENSION

Set this keyword equal to a vector of 1 to 8 elements specifying the dimensions of the result. Setting this keyword is equivalent to specifying an array via the *D* argument.

DOUBLE

Set this keyword to specify that the memory segment should be treated as a double-precision floating-point array.

FILENAME

By default, SHMMAP maps anonymous shared memory. Set the FILENAME keyword equal to a string containing the path name of a file to be mapped to create a memory-mapped file. A shared mapped file can serve as shared memory between unrelated processes. The primary difference between anonymous shared memory and mapped files is that mapped files require a file of the specified size to exist in the filesystem, whereas anonymous shared memory has no user-visible representation in the filesystem.

Unless the PRIVATE keyword is also specified, changes made to such a mapped file are written back to the file by the operating system, and are visible to any other process that is mapping the same file.

Note

The non-private form of file mapping corresponds to the `MAP_SHARED` flag to the UNIX `mmap()` function, or the `PAGE_READWRITE` to the Windows `CreateFileMapping()` system call.

FLOAT

Set this keyword to specify that the memory segment should be treated as a single-precision floating-point array.

GET_NAME

If *SegmentName* is not specified in a call to SHMMAP, IDL automatically generates a name. Set this keyword equal to a named variable that will receive the name assigned by IDL to the memory segment.

GET_OS_HANDLE

Set this keyword equal to a named variable that will receive the operating system name (or *handle*) for the memory segment. The meaning of the operating system handle depends on both the operating system and the type of memory segment used. See the description of the OS_HANDLE keyword for details.

INTEGER

Set this keyword to specify that the memory segment should be treated as an integer array.

L64

Set this keyword to specify that the memory segment should be treated as a 64-bit integer array.

LONG

Set this keyword to specify that the memory segment should be treated as a longword integer array.

OFFSET

If present and non-zero, this keyword specifies an offset (in bytes) from the start of the shared memory segment or memory mapped file that will be used as the base address for the IDL array associated with the memory segment.

Note

Most computer hardware is not able to access arbitrary data types at arbitrary memory addresses. Data must be properly aligned for its type or the program will crash with an alignment error (often called a *bus error*) when the data is accessed. The specific rules differ between machines, but in many cases the address of a data object must be evenly divisible by the size of that object. IDL will issue an error if you specify an offset that is not valid for the array specified.

Note

The actual memory mapping primitives provided by the underlying operating system require such offsets to be integer multiples of the virtual memory pagesize (sometimes called the *allocation granularity*) for the system. This value is typically a power of two such as 8K or 64K. In contrast, IDL allows arbitrary offsets as long as they satisfy the alignment constraints of the data type. This is implemented by mapping the page that contains the specified offset, and then adjusting the memory address to point at the specified byte within that page. In rounding your offset request back to the nearest page boundary, IDL may map slightly more memory than your request would seem to require, but never more than a single page.

OS_HANDLE

Set this keyword equal to the name (or *handle*) used by the underlying operating system for the memory segment. If you do not specify the OS_HANDLE keyword, SHMMAP will under some circumstances provide a default value. The specific meaning and syntax of the OS_HANDLE depends on both the operating system and the form of memory used. See the following sections for operating-system specific behavior, and "Types Of Memory Segments" on page 430 for behavior differences based on the form of memory used.

Posix (UNIX) Shared Memory

Use the OS_HANDLE keyword to supply a string value containing the system global name of the shared memory segment. Such names are expected to start with a slash `(/)` character, and not to contain any other slash characters. You can think of this as mimicking the syntax for a file in the root directory of the system, although no such file is created. See your system documentation for the `shm_open()` system call for specific details. If you do not supply the OS_HANDLE keyword, SHMMAP will create one for you by prepending a slash character to the value given by the *SegmentName* argument.

UNIX System V Shared Memory

Use the OS_HANDLE keyword to supply an integer value containing the system global identifier of an existing shared memory segment to attach to the process. If you do not supply the OS_HANDLE keyword, then SHMMAP creates a new memory segment. The identifier for this segment is available via the GET_OS_HANDLE keyword.

Windows Anonymous Shared Memory

Use the OS_HANDLE keyword to supply a global system name for the mapping object underlying the anonymous shared memory. If the OS_HANDLE keyword is not specified, SHMMAP uses the value of the *SegmentName* argument.

UNIX Memory Mapped Files

The OS_HANDLE keyword has no meaning for UNIX memory mapped files and is quietly ignored.

Windows Memory Mapped Files

Use the OS_HANDLE keyword to supply a global system name for the mapping object underlying the mapped file. Use of the OS_HANDLE will ensure that every process accessing the shared file will see a coherent view of its contents, and is thus recommended for Windows memory mapped files. However, if you do not supply the

OS_HANDLE handle keyword for a memory mapped file, no global name is passed to the Windows operating system, and a unique mapping object for the file will be created.

PRIVATE

Set this keyword to specify that a private file mapping is required. In a private file mapping, any changes written to the mapped memory are visible only to the process that makes them, and such changes are not written back to the file. This keyword is ignored unless the FILENAME keyword is also present.

Note

Due to limitations of the operating system, the PRIVATE keyword is not allowed under the Windows 9x operating systems (Windows 95, Windows 98, Windows ME). Windows NT and related systems do not have this limitation.

Note

Under UNIX, the private form of file mapping corresponds to the `MAP_PRIVATE` flag to the `mmap()` system call. Under Windows, the non-private form corresponds to the `PAGE_WRITECOPY` option to the Windows `CreateFileMapping()` system call. When your process alters data within a page of privately mapped memory, the operating system performs a *copy on write* operation in which the contents of that page are copied to a new memory page visible only to your process. This private memory usually comes from anonymous swap space or the system pagefile. Hence, private mapped files require more system resources than shared mappings.

It is possible for some processes to use private mappings to a given file while others use a public mapping to the same file. In such cases, the private mappings will see changes made by the public processes up until the moment the private process itself makes a change to the page. The pagesize granularity and timing issues between such processes can make such scenarios very difficult to control. RSI does not recommend combining simultaneous shared and private mappings to the same file.

SIZE

Set this keyword equal to a `size` vector specifying the type and dimensions to be associated with the memory segment. The format of a size vector is given in the description of the SIZE function.

SYSV

Under UNIX, the default form of anonymous memory is Posix shared memory, (`shm_open()` and `shm_unlink()`). Specify the SYSV keyword to use System V shared memory (`shmget()`, `shmctl()`, and `shmdt()`) instead. On systems where it is available, Posix shared memory is more flexible and has fewer limitations. System V shared memory is available on all UNIX implementations, and serves as an alternative when Posix memory does not exist, or when interfacing to exiting non-IDL software that uses System V shared memory. See "Types Of Memory Segments" on page 430 for a full discussion.

TEMPLATE

Set this keyword equal to a variable of the type and dimensions to be associated with the memory segment.

TYPE

Set this keyword to specify the type code for the memory segment. See the description of the SIZE function for a list of IDL type codes.

UINT

Set this keyword to specify that the memory segment should be treated as a unsigned integer array.

ULONG

Set this keyword to specify that the memory segment should be treated as a unsigned longword integer array.

UL64

Set this keyword to specify that the memory segment should be treated as a unsigned 64-bit integer array.

Types Of Memory Segments

SHMMAP is a relatively direct interface to the shared memory and file mapping primitives provided by the underlying operating system. The SHMMAP interface attempts to minimize the differences between these primitives, and for simple shared memory use, it may not be necessary to fully understand the underlying mechanisms. For most purposes, however, it is necessary to understand the operating system primitives in order to understand how to use SHMMAP properly.

UNIX

In modern UNIX systems, the `mmap()` system call forms the primary basis for both file mapping and anonymous shared memory. The existence of System V shared memory, which is an older form of anonymous shared memory, adds some complexity to the situation.

UNIX Memory Mapped Files

To memory map a file under UNIX, you open the file using the `open()` system call, and then map it using `mmap()`. Once the file is mapped, you can close the file, and the mapping remains in place until explicitly unmapped, or until the process exits or calls `exec()` to run a different program.

If more than one process maps a file at the same time using the `MAP_SHARED` flag to `mmap()`, then those processes will be able to see each others' changes. Hence, memory mapped files are one form of shared memory. Although the requirement for a scratch file large enough to satisfy the mapping is inconvenient, limitations in System V shared memory have led many UNIX programmers to use memory mapped files in this way.

UNIX System V Shared Memory

Anonymous shared memory has traditionally been implemented via an API commonly referred to as System V IPC. The `shmget()` function is used to create a shared memory segment. The caller does not name the segment. Instead, the operating system assigns each such segment a unique integer ID when it is created. Once a shared memory segment exists, the `shmdt()` function can be used to map it into the address space of any process that knows the identifier. This segment persists in the OS kernel until it is explicitly destroyed via the `shmctl()` function, or until the system is rebooted. This is true even if there are no processes currently mapped to the segment. This can be convenient in situations where the need for the shared memory is long lived, and programs that need it come and go. It also can create a problem, however, since shared memory segments that are not explicitly destroyed can cause memory leaks in the operating system. Hence, it is important to properly destroy such segments when they are no longer required.

System V shared memory has been part of UNIX for a long time. It is available on all UNIX platforms, and there is a large amount of existing code that uses it. There are, however, some limitations on its utility:

- Many systems place extremely small limits on the size allowed for such memory segments. These limits are often kernel parameters that can be adjusted by the system administrator. The details are highly system dependent. Consult your system documentation for details.

- The caller does not have the option of naming the shared memory segment. Instead, the operating system assigns an arbitrary number, which means that processes that want to map such a segment have to have a mechanism for finding the correct identifier to use before they can proceed. This, in turn, requires some additional form of interprocess communication.

RSI recommends the use of Posix shared memory instead of System V shared memory for those platforms that support it and applications that can use it. Under UNIX, SHMMAP defaults to Posix shared memory to implement anonymous shared memory. To use System V shared memory, you must specify the SYSV keyword. See the Examples section below for an example of using System V shared memory.

Posix Shared Memory

Posix shared memory is a newer alternative for anonymous shared memory. It is part of the UNIX98 standard, and although not all current UNIX systems support it, it will in time be available on all UNIX systems. Posix shared memory uses the `shm_open()` and `ftruncate()` system calls to create a memory segment that can be accessed via a file descriptor. This descriptor is then used with the `mmap()` system call to map the memory segment in the usual manner. The primary difference between this, and simply using `mmap()` on a scratch file to implement shared memory is that no scratch file is required (the disk space comes from the system swapspace). As with System V shared memory, Posix shared memory segments exist in the operating system until explicitly destroyed (using the `shm_unlink()` system call). Unlike System V shared memory, but like all the other forms, Posix shared memory allows the caller to supply the name of the segment. This simplifies the situation in which multiple processes want to map the same segment. One of them creates it, and the others simply map it, all of them using the same name to reference it.

Posix shared memory is the default for SHMMAP on all UNIX platforms — even those that do not yet support it. (To use System V shared memory instead, you must specify the SYSV keyword.) There are several reasons for making Posix shared memory the default for all UNIX platforms:

- To remain UNIX compliant, all platforms will have to implement the UNIX98 standard. Most have, and the remainder are currently in the process of doing so. We believe that Posix shared memory will be available on all UNIX systems very soon.
- Having different defaults for different UNIX platforms would cause unnecessary confusion; the confusion would only increase as platforms added support for Posix shared memory, causing the platform’s SHMMAP default to change with later IDL releases. Since in most cases you need to know the

underlying mechanism in use, the default should be easy to determine, and should not change over time.

- In the long run, it is desirable for the best option to be the default.

Microsoft Windows

Under Microsoft Windows, the `CreateFileMapping()` system call forms the basis for shared memory as well as memory mapped files. To map a file, you open the file and then pass the handle for that file to `CreateFileMapping()`. To create a region of anonymous mapped memory instead of a mapped file, you pass a special file handle (`0xffffffff`) to `CreateFileMapping()`. In this case, the disk space used to back the shared memory is taken from the system pagefile. `CreateFileMapping()` accepts an optional parameter (`lpname`), which if present, is used to give the resulting memory mapping object a system global name. If you specify such a name, and a mapping object with that name already exists, you will receive a handle to the existing mapping object. Otherwise, `CreateFileMapping()` creates a new mapping object for the file. Hence, to create anonymous (no file) shared memory between unrelated processes, IDL calls `CreateFileMapping()` with the special `0xffffffff` file handle, and specifies a global name for it.

A global name (supplied via the OS_HANDLE keyword) is the only name by which an anonymous shared memory segment can be referenced within the system. Global names are not required for memory mapped files, because each process can create a separate mapping object and use it to refer to the same file. Although this does allow the unrelated processes to see each others' changes, their views of the file will not be *coherent* (that is, identical). With coherent access, all processes see exactly the same memory at exactly the same time because they are all mapping the same physical page of memory. To get coherent access to a memory mapped file, every process should specify the OS_HANDLE keyword to ensure that they use the same mapping object. Coherence is only an issue when the contents of the file are altered; when using read-only access to a mapped file, you need not be concerned with this issue.

The Windows operating system automatically destroys a mapping object when the last process with an open handle to it closes that handle. Destruction of the mapping object may be the result of an explicit call to `CloseHandle()`, or may involve an implicit close that happens when the process exits. This differs from the UNIX behavior for anonymous shared memory, and consequently the benefits and disadvantages are reversed. The advantage is that it is not possible to forget to destroy a mapping object, and end up with the operating system holding memory that is no longer useful, but which cannot be freed. On the other hand, you must ensure that at least one open handle to the object is open at all times, or the system might free an object that you intended to use again.

Note

Under Windows, when attaching to an existing memory object by providing the global segment name, IDL is not able to verify that the memory segment returned by the operating system is large enough to satisfy the IDL array specified to SHMMAP for its type and size. If the segment is not large enough, the IDL program will crash with an illegal memory access exception when it attempts to access memory addresses beyond the end of the segment. Hence, the IDL user must ensure that such pre-existing memory segments are long enough for the specified IDL array.

Reference Counts And Memory Segment Lifecycle

You can see a list of all current memory segments created with SHMMAP by issuing the statement

```
HELP,/SHARED_MEMORY
```

To access a current segment, it must be tied to an IDL variable using the SHMVAR function. IDL maintains a reference count of the number of variables currently accessing each memory segment, and does not allow a memory segment to be removed from the IDL process as long as variables that reference it still exist.

SHMMAP will not allow you to create a new memory segment with the same *SegmentName* as an existing segment. You should therefore be careful to pick unique segment names. One way to ensure that segment names are unique is to not provide the *SegmentName* argument when calling SHMMAP. In this case, SHMMAP will automatically choose a unique name, which can be obtained using the GET_NAME keyword.

The SHMUNMAP procedure is used to remove a memory segment from the IDL session. In addition, it may remove the memory segment from the system. (Whether the memory segment is removed from the system depends on the type of segment, and on the arguments used with SHMMAP when the segment was initially attached.) If no variables from the current IDL session are accessing the segment (that is, if the IDL-maintained reference count is 0), the segment is removed immediately. If variables in the current IDL session are still referencing the segment, the segment is marked for removal when the last such variable drops its reference. Once SHMMAP is called on a memory segment, no additional calls to SHMVAR are allowed for it within the current IDL session; this means that a segment marked by SHMUNMAP as *UnmapPending* cannot be used for new variables within the current IDL session.

Note

IDL has no way to determine whether a process other than itself is accessing a shared memory segment. As a result, it is possible for IDL to destroy a memory segment that is in use by another process. The specific details depend on the type of memory segment, and the options used with SHMMAP when the segment was loaded. When creating applications that use shared memory, you should ensure that all applications that use the segment (be they instances of IDL or any other application) communicate regarding their use of the shared memory and act in a manner that avoids this pitfall.

Examples

Example 1

Create a shared memory segment of 1000000 double-precision data elements, and then fill it with a DINDGEN ramp:

```
SHMMAP, 'MYSEG', /DOUBLE, 1000000
z = SHMVAR('MYSEG')
z[0] = DINDGEN(1000000)
```

Note

When using shared memory, using the explicit subscript of the variable (z, in this case) maintains the variable’s connection with the shared memory segment. When not using shared memory, assignment without subscripting is more efficient and is recommended.

Example 2

Create the same shared memory segment as the previous example, but let IDL choose the segment name:

```
SHMMAP, /DOUBLE, DIMENSION=[1000000], GET_NAME=segname
z = SHMVAR(segname)
z[0] = DINDGEN(1000000)
```

Example 3

Create the same shared memory segment as the previous example, but use a temporary file, mapped into IDL’s address space, instead of anonymous shared memory. The file needs to be the correct length for the data we will be mapping onto it. We satisfy this need while simultaneously initializing it with the DINDGEN vector by writing the vector to the file.

The use of the OS_HANDLE keyword improves performance and correctness under Windows while being quietly ignored under UNIX:

```
filename = FILEPATH('idl_scratch', /TMP)
OPENW, unit, filename, /GET_LUN
WRITEU, unit, DINDGEN(1000000)
CLOSE, unit
SHMMAP, /DOUBLE, DIMENSION=[1000000], GET_NAME=segname, $
FILENAME=filename, OS_HANDLE='idl_scratch'
z = SHMVAR(segname)
```

Example 4

Create an anonymous shared memory segment using UNIX System V shared memory. Use of System V shared memory differs from the other methods in two ways:

- The system identifier for the segment is a number chosen by the system instead of a name selected by the user.
- With SYSV memory, you have to explicitly indicate whether the operation is a create operation (no OS_HANDLE keyword) or merely an attach to an existing segment (OS_HANDLE is present). The other methods create the segment as needed, and will automatically attach to a memory segment with the desired operating system handle if it already exists. The SHMMAP call does not explicitly have to specify that the segment should be created.

In this example, we will use the type and size of the existing `myvar` variable to determine the size of the memory:

```
SHMMAP, TEMPLATE=myvar, GET_NAME=segname, /SYSV, $
   GET_OS_HANDLE=oshandle
```

In this case, the SYSV keyword forces the use of System V shared memory. The absence of the OS_HANDLE keyword tells SHMMAP to create the segment, instead of simply mapping an existing one. In a different IDL session running on the same machine, if you knew the proper OS_HANDLE value for this segment, you could attach to the segment created above as follows:

```
SHMMAP, TEMPLATE=myvar, GET_NAME=segname, /SYSV, $
   OS_HANDLE=oshandle
```

In this case, the OS_HANDLE keyword tells SHMMAP the identifier of the memory segment, causing it to attach to the existing segment instead of creating a new one.

Version History

Introduced: 5.6

See Also

SHMDEBUG, SHMUNMAP, SHMVAR

SHMUNMAP

The SHMUNMAP procedure is used to remove a memory segment previously created by SHMMAP from the IDL session. In addition, it may remove the memory segment from the system. (Whether the memory segment is removed from the system depends on the type of segment, and on the arguments used with SHMMAP when the segment was initially attached.) If no variables from the current IDL session are accessing the segment (that is, if the IDL-maintained reference count is 0), the segment is removed immediately. If variables in the current IDL session are still referencing the segment, the segment is marked for removal when the last such variable drops its reference.

During this *UnmapPending* phase:

- The segment still exists in the system, so attempts to use SHMMAP to create a new segment with the same *SegmentName* will fail.
- Additional calls to SHMVAR to attach new variables to this segment will fail.

Note

IDL has no way to determine whether a process other than itself is accessing a shared memory segment. As a result, it is possible for IDL to destroy a memory segment that is in use by another process. The specific details depend on the type of memory segment, and the options used with SHMMAP when the segment was loaded. When creating applications that use shared memory, you should ensure that all applications that use the segment (be they instances of IDL or any other application) communicate regarding their use of the shared memory and act in a manner that avoids this pitfall.

Syntax

SHMUNMAP, *SegmentName*

Arguments

SegmentName

A scalar string containing the IDL name for the shared memory segment, as assigned by SHMMAP.

Examples

To destroy a memory segment previously created by SHMMAP with the segment name `myseg`:

```
SHMUNMAP, 'myseg'
```

Version History

Introduced: 5.6

See Also

SHMDEBUG, SHMMAP, SHMVAR

SHMVAR

The SHMVAR function creates an IDL array variable that uses the memory from a current mapped memory segment created by the SHMMAP procedure. Variables created by SHMVAR are used in much the same way as any other IDL variable, and provide the IDL user with the ability to alter the contents of anonymous shared memory or memory mapped files.

By default, the variable created by SHMVAR is given the type and dimensions that were specified to SHMMAP when the memory segment was created. However, this default can be changed by SHMVAR via a variety of keywords as well as via the D_i arguments. The created array can be of any type except for pointer, object reference, or string. Structure types are allowed as long as they do not contain any pointers, object references, or strings.

Syntax

Result = SHMVAR(*SegmentName* [, D_1, ..., D_8] [, /BYTE] [, /COMPLEX] [, /DCOMPLEX] [, DIMENSION=*value*] [, /DOUBLE] [, /FLOAT] [, /INTEGER] [, /L64] [, /LONG] [, SIZE=*value*] [, TEMPLATE=*value*] [, TYPE=*value*] [, /UINT] [, /UL64] [, /ULONG])

Return Value

An IDL array variable that uses memory from a the specified mapped memory segment.

Arguments

SegmentName

A scalar string supplying the IDL name for the shared memory segment, as assigned by SHMMAP.

D_i

The dimensions of the result. The D_i arguments can be either a single array containing the dimensions or a sequence of scalar dimensions. Up to eight dimensions can be specified. If no dimensions are specified, the parameters specified to SHMMAP are used.

Keywords

BYTE

Set this keyword to specify that the memory segment should be treated as a byte array.

COMPLEX

Set this keyword to specify that the memory segment should be treated as a complex, single-precision floating-point array.

DCOMPLEX

Set this keyword to specify that the memory segment should be treated as a complex, double-precision floating-point array.

DIMENSION

Set this keyword equal to a vector of 1 to 8 elements specifying the dimensions of the result. This is equivalent to the array form of the D_i plain arguments. If no dimensions are specified, the parameters specified to SHMMAP are used.

DOUBLE

Set this keyword to specify that the memory segment should be treated as a double-precision floating-point array.

FLOAT

Set this keyword to specify that the memory segment should be treated as a single-precision floating-point array.

INTEGER

Set this keyword to specify that the memory segment should be treated as an integer array.

L64

Set this keyword to specify that the memory segment should be treated as a 64-bit integer array.

LONG

Set this keyword to specify that the memory segment should be treated as a longword integer array.

SIZE

Set this keyword equal to a size vector specifying the type and dimensions to be associated with the memory segment. The format of a size vector is given in the description of the SIZE function. If no dimensions are specified, the parameters specified to SHMMAP are used.

TEMPLATE

Set this keyword equal to a variable of the type and dimensions to be associated with the memory segment. If no dimensions are specified, the parameters specified to SHMMAP are used.

TYPE

Set this keyword to specify the type code for the memory segment. See the description of the SIZE function for a list of IDL type codes.

UINT

Set this keyword to specify that the memory segment should be treated as a unsigned integer array.

ULONG

Set this keyword to specify that the memory segment should be treated as a unsigned longword integer array.

UL64

Set this keyword to specify that the memory segment should be treated as a unsigned 64-bit integer array.

Examples

See the examples given for the SHMMAP procedure.

Version History

Introduced: 5.6

See Also

SHMDEBUG, SHMMAP, SHMUNMAP

SKIP_LUN

The SKIP_LUN procedure reads data in an open file and moves the file pointer. It is useful in situations where it is necessary to skip over a known amount of data in a file without the requirement of having the data available in an IDL variable. SKIP_LUN can skip over a fixed amount of data, specified in bytes or lines of text, or can skip over the remainder of the input file from the current position to end of file. Since SKIP_LUN actually performs an input operation to advance the file pointer, it is not as efficient as POINT_LUN for skipping over a fixed number of bytes in a disk file. For that reason, use of POINT_LUN is preferred when possible. SKIP_LUN is especially useful in situations such as:

- Skipping over a fixed number of lines of text. Since lines of text can have variable length, it can be difficult to use POINT_LUN to skip them.
- Skipping data from a file that is not a regular disk file (for example, data from an internet socket).

Syntax

SKIP_LUN, *FromUnit*, [, *Num*] [, /EOF] [, /LINES]
[, /TRANSFER_COUNT=*variable*]

Arguments

FromUnit

An integer that specifies the file unit for the file in which the file pointer is to be moved. Data in *FromUnit* is skipped, starting at the current position of the file pointer. The file pointer is advanced as data is read and skipped. The file specified by *FromUnit* must be open, and must not have been opened with the RAWIO keyword to OPEN.

Num

The amount of data to skip. This value is specified in bytes, unless the LINES keyword is specified, in which case it is taken to be the number of text lines. If *Num* is not specified, SKIP_LUN acts as if the EOF keyword has been set, and skips all data in *FromUnit* (the source file) from the current position of the file pointer to the end of the file.

If *Num* is specified and the source file comes to end of file before the specified amount of data is skipped, SKIP_LUN issues an end-of-file error. The EOF keyword alters this behavior.

Keywords

EOF

Set this keyword to ignore the value given by *Num* (if present) and instead skip all data from the current position of the file pointer in *FromUnit* and the end of the file.

Note

If EOF is set, no end-of-file error is issued even if the amount of data skipped does not match the amount specified by *Num*. The TRANSFER_COUNT keyword can be used with EOF to determine how much data was skipped.

LINES

Set this keyword to indicate that the *Num* argument specifies the number of lines of text to be skipped. By default, the *Num* argument specifies the number of bytes of data to skip.

TRANSFER_COUNT

Set this keyword equal to a named variable that will contain the amount of data skipped. If LINES is specified, this value is the number of lines of text. Otherwise, it is the number of bytes. TRANSFER_COUNT is primarily useful in conjunction with the EOF keyword. If EOF is not specified, TRANSFER_COUNT will be the same as the value specified for *Num*.

Examples

Skip the next 8 lines of text from a file:

```
SKIP_LUN, FromUnit, 8, /LINES
```

Skip the remainder of the data in a file, and use the TRANSFER_COUNT keyword to determine how much data was skipped:

```
SKIP_LUN, FromUnit, /EOF, TRANSFER_COUNT=n
```

Skip the remainder of the text lines in a file, and use the TRANSFER_COUNT keyword to determine how many lines were skipped:

```
SKIP_LUN, FromUnit, /EOF, /LINES, TRANSFER_COUNT=n
```

Version History

Introduced: 5.6

See Also

CLOSE, COPY_LUN, EOF, FILE_COPY, FILE_LINK, FILE_MOVE, OPEN, POINT_LUN, PRINT/PRINTF, READ/READF, WRITEU

SWAP_ENDIAN_INPLACE

The SWAP_ENDIAN_INPLACE procedure reverses the byte ordering of arbitrary scalars, arrays or structures. It can make “big endian” number “little endian” and vice-versa.

Note

The BYTEORDER procedure can be used to reverse the byte ordering of *scalars and arrays* (SWAP_ENDIAN_INPLACE also allows structures).

SWAP_ENDIAN_INPLACE differs from the SWAP_ENDIAN function in that it alters the input data in place rather than making a copy as does SWAP_ENDIAN. SWAP_ENDIAN_INPLACE can therefore be more efficient, if a copy of the data is not needed. The pertinent bytes in the input variable are reversed.

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

Syntax

SWAP_ENDIAN_INPLACE, *Variable* [, /SWAP_IF_BIG_ENDIAN]
[, /SWAP_IF_LITTLE_ENDIAN]

Arguments

Variable

The named variable—scalar, array, or structure—to be swapped.

Keywords

SWAP_IF_BIG_ENDIAN

If this keyword is set, the swap request will only be performed if the platform running IDL uses “big endian” byte ordering. On little endian machines, the SWAP_ENDIAN_INPLACE request quietly returns without doing anything. Note that this keyword does not refer to the byte ordering of the input data, but to the computer hardware.

SWAP_IF_LITTLE_ENDIAN

If this keyword is set, the swap request will only be performed if the platform running IDL uses “little endian” byte ordering. On big endian machines, the SWAP_ENDIAN_INPLACE request quietly returns without doing anything. Note that this keyword does not refer to the byte ordering of the input data, but to the computer hardware.

Examples

Reverse the byte order of A:

```
SWAP_ENDIAN_INPLACE, A
```

Version History

Introduced: 5.6

See Also

BYTEORDER, SWAP_ENDIAN

TRUNCATE_LUN

The TRUNCATE_LUN procedure truncates the contents of a file (which must be open for write access) at the current position of the file pointer. After this operation, all data before the current file pointer remains intact, and all data following the file pointer are gone. The position of the current file pointer is not altered.

Syntax

TRUNCATE_LUN, $Unit_1$, ..., $Unit_n$

Arguments

$Unit_n$

Scalar or array variables containing the logical file unit numbers of the open files to be truncated.

Keywords

None.

Examples

Example 1

Truncate the entire contents of an existing file:

```
OPENU, unit, 'baddata.dat', /GET_LUN
TRUNCATE_LUN, unit
FREE_LUN, unit
```

Example 2

Given an existing file of 10,000 bytes, throw away the final 5,000 bytes, and then write an additional 2,000 byte array in their place. The resulting file will be 7,000 bytes in length.

```
OPENU, unit, 'mydata.dat', /GET_LUN
POINT_LUN, unit, 5000
TRUNCATE_LUN, unit
WRITEU, unit, BYTARR(2000)
FREE_LUN, unit
```

Version History

Introduced: 5.6

See Also

GET_LUN, OPEN, POINT_LUN

WIDGET_COMBOBOX

The WIDGET_COMBOBOX function creates combobox widgets, which are similar to droplist widgets. The main difference between the combobox widget and the droplist widget is that the combobox widget can be created in such a way that the text field is editable, allowing the user to enter a value that is not on the list.

A combobox widget displays a text field and an arrow button. If the combobox is not editable, selecting either the text field or the button reveals a list of options from which to choose. When the user selects a new option from the list, the list disappears and the text field displays the currently-selected option. This action generates an event containing the index of the selected item, which ranges from zero to the number of elements in the list minus one.

If the combobox is editable, text can be entered in the text box without causing the list to drop down. This action causes an event in which the index field is set to -1, allowing you to distinguish this event from list selections.

The text of the current selection is returned in the STR field of the WIDGET_COMBOBOX event structure. See "Widget Events Returned by Combobox Widgets" on page 458 for details.

Note

WIDGET_COMBOBOX is not currently available on Compaq True64 UNIX platforms due to that platform's lack of support for the necessary Motif libraries.

Syntax

Result = WIDGET_COMBOBOX(*Parent* [, /DYNAMIC_RESIZE] [, /EDITABLE] [, EVENT_FUNC=*string*] [, EVENT_PRO=*string*] [, FONT=*string*] [, FRAME=*value*] [, FUNC_GET_VALUE=*string*] [, GROUP_LEADER=*widget_id*] [, KILL_NOTIFY=*string*] [, /NO_COPY] [, NOTIFY_REALIZE=*string*] [, PRO_SET_VALUE=*string*] [, RESOURCE_NAME=*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 combobox widget.

Arguments

Parent

The widget ID of the parent widget for the new combobox widget.

Keywords

DYNAMIC_RESIZE

Set this keyword to create a widget that resizes itself to fit its new value whenever its value is changed.

Note

This keyword does not take effect when used with the SCR_XSIZE, SCR_YSIZE, XSIZE, or YSIZE keywords. If one of these keywords is also set, the widget will be sized as specified by the sizing keyword and will never resize itself dynamically.

EDITABLE

Set this keyword to create an editable combobox. If the combobox is editable, users can enter or modify in the text field. Changes in the combobox text field will cause combobox events with the INDEX field of the event structure set to -1. The current text will be ‘ in the STR field of the event structure.

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” on page 3938 in the *IDL Reference Guide* 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.

FRAME

The value of this keyword specifies the width of a frame in units specified by the UNITS keyword (pixels are the default) to be drawn around the borders of the widget.

Note

This keyword is only a hint to the toolkit, and may be ignored in some instances.

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.

GROUP_LEADER

The widget ID of an existing widget that serves as group leader for the newly-created widget. When a group leader is killed, for any reason, all widgets in the group are also destroyed.

A given widget can be in more than one group. The WIDGET_CONTROL procedure can be used to add additional group associations to a widget. You cannot remove a widget from an existing group.

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. This callback procedure can be removed by setting the routine name 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.

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 works well 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 copying 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. Upon a set operation (using the UVALUE keyword to WIDGET_COMBOBOX or the SET_UVALUE keyword to WIDGET_CONTROL), the variable passed as value becomes undefined. Upon 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 containing 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 callback procedure. This callback procedure can be removed by setting the routine name 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 to the user.

RESOURCE_NAME

A string containing an X Window System resource name to be applied to the widget. See "RESOURCE_NAME" on page 2055 in the *IDL Reference Guide* for a complete discussion of this keyword.

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.

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" on page 2061 in the documentation for WIDGET_BASE in the *IDL Reference Guide*.

UNAME

Set this keyword to a string, which is 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 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.

VALUE

The initial value setting of the widget. The value of a combobox widget is a scalar string or array of strings that contains the text of the list items (one list item per array element). Combobox widgets are sized based on the length (in characters) of the longest item specified in the array of values for the VALUE keyword.

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 plain base widget. You should avoid using this style of widget programming.

XSIZE

The desired width of the combobox widget area, in units specified by the UNITS keyword (pixels are the default). Most widgets attempt to size themselves to fit the situation. However, if the desired effect is not produced, use this keyword to override it. This keyword does not control the size of the combobox button or of the dropped list. Instead, it controls the size around the combobox button and, as such, is not particularly useful.

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 plain base widget. You should avoid using this style of widget programming.

YSIZE

The desired height of the widget, in units specified by the UNITS keyword (pixels are the default). Most widgets attempt to size themselves to fit the situation. However, if the desired effect is not produced, use this keyword to override it. This keyword does not control the size of the combobox button or of the dropped list. Instead, it controls the size around the combobox button and, as such, is not particularly useful.

Keywords to WIDGET_CONTROL

A number of keywords to the WIDGET_CONTROL affect the behavior of combobox widgets. In addition to those keywords that affect all widgets, the following keywords are particularly useful: COMBOBOX_ADDITEM, COMBOBOX_DELETEITEM, COMBOBOX_INDEX, DYNAMIC_RESIZE, GET_VALUE, SET_COMBOBOX_SELECT, SET_VALUE.

Keywords to WIDGET_INFO

A number of keywords to the WIDGET_INFO return information that applies specifically to combobox widgets. In addition to those keywords that apply to all widgets, the following keywords are particularly useful: COMBOBOX_GETTEXT, COMBOBOX_NUMBER, DYNAMIC_RESIZE.

Widget Events Returned by Combobox Widgets

Pressing the mouse button while the mouse pointer is over an element of a combobox widget causes the widget to change the text field on the combobox and to generate an event. The event structure returned by the WIDGET_EVENT function is defined by the following statement:

```
{WIDGET_COMBOBOX, ID:0L, TOP:0L, HANDLER:0L, INDEX:0L, STR:""}
```

The first three fields are the standard fields found in every widget event. INDEX returns the index of the selected item. This can be used to index the array of names originally used to set the widget's value. If the event was caused by text changes in an editable combobox, the INDEX field will be set to -1. If you are using an editable combobox, it is important to check for the value of -1 prior to using the value of the INDEX field as an index into the array if items. The text of the current selection is returned in the STR field, which may eliminate the need to use the index field in many cases.

Note

Platform-specific UI toolkits behave differently if a combobox widget has only a single element. On some platforms, selecting that element again does not generate an event. Events are always generated if the list contains multiple items.

Version History

Introduced: 5.6

See Also

CW_PDMENU, WIDGET_BUTTON, WIDGET_DROPLIST, WIDGET_LIST

WIDGET_TAB

The WIDGET_TAB function is used to create a tab widget. Tab widgets present a display area on which different pages (base widgets and their children) can be displayed by selecting the appropriate tab. The titles of the tabs are supplied as the values of the TITLE keyword for each of the tag widget's child base widgets.

For a more detailed discussion of the tab widget, along with examples, see "Using Tab Widgets" in Chapter 26 of the *Building IDL Applications* manual.

Syntax

Result = WIDGET_TAB(*Parent* [, /ALIGN_BOTTOM | , /ALIGN_CENTER | , /ALIGN_LEFT | , /ALIGN_RIGHT | , /ALIGN_TOP] [, EVENT_FUNC=*string*] [, EVENT_PRO=*string*] [, FUNC_GET_VALUE=*string*] [, GROUP_LEADER=*widget_id*] [, KILL_NOTIFY=*string*] [, LOCATION={0 | 1 | 2 | 3}] [, MULTILINE=0 | 1 (Windows) or *num* tabs per row (Motif)] [, /NO_COPY] [, NOTIFY_REALIZE=*string*] [, PRO_SET_VALUE=*string*] [, SCR_XSIZE=*width*] [, SCR_YSIZE=*height*] [, /SENSITIVE] [, UNAME=*string*] [, UNITS={0 | 1 | 2}] [, UVALUE=*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 tab widget.

Arguments

Parent

The widget ID of the parent for the new tab widget.

Note
Only base widgets can be the parent of a 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.

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.

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.

GROUP_LEADER

The widget ID of an existing widget that serves as group leader for the newly-created widget. When a group leader is killed, for any reason, all widgets in the group are also destroyed.

A given widget can be in more than one group. The WIDGET_CONTROL procedure can be used to add additional group associations to a widget. You cannot remove a widget from an existing group.

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. This callback procedure can be removed by setting the routine name to the null string (`''`). Note that the procedure specified is used only if you are not using the XMANAGER procedure to manage your widgets.

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.

If you use the XMANAGER procedure to manage your widgets, the value of this keyword is overwritten. Use the CLEANUP keyword to XMANAGER to specify a procedure to be called when a managed widget dies.

LOCATION

Set this keyword equal to an integer that specifies which edge of the tab widget will contain the tabs. The possible values are:

Value	Description
0	The tabs are placed along the top of the widget, which is the default behavior.
1	The tabs are placed along the bottom of the widget.

Table 3-33: LOCATION Keyword Values

Value	Description
2	The tabs are placed along the left edge of the widget. The text label for each tab is displayed vertically. On Windows platforms, setting the keyword to this value implies the MULTILINE keyword.
3	The tabs are placed along the right edge of the widget. The text label for each tab is displayed vertically. On Windows platforms, setting the keyword to this value implies the MULTILINE keyword.

Table 3-33: LOCATION Keyword Values (Continued)

MULTILINE

This keyword controls how tabs appear on the tab widget when all of the tabs do not fit on the widget in a single row. This keyword behaves differently on Windows and Motif systems.

Windows

Set this keyword to cause tabs to be organized in a multiline display when the width of the tabs exceeds the width of the largest child base widget. If possible, IDL will create tabs that display the full tab text.

If MULTILINE = 0 and LOCATION = 0 or 1, tabs that exceed the width of the largest child base widget are shown with scroll buttons, allowing the user to scroll through the tabs while the base widget stays immobile.

If LOCATION = 1 or 2, a multiline display is always used if the tabs exceed the height of the largest child base widget.

Note

The width or height of the tab widget is based on the width or height of the largest base widget that is a child of the tab widget. The text of the tabs (the titles of the tab widget's child base widgets) may be truncated even if the MULTILINE keyword is set.

Motif

Set this keyword equal to an integer that specifies the maximum number of tabs to display per row in the tab widget. If this keyword is not specified (or is explicitly set equal to zero) all tabs are placed in a single row.

Note

The width or height of the tab widget is based on the width or height of the largest base widget that is a child of the tab widget. The text of the tabs (the titles of the tab widget's child base widgets) is never truncated in order to make the tabs fit the space available. However, tab text may be truncated if the text of a single tab exceeds the space available. If MULTILINE is set to any value other than one, some tabs may not be displayed.

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 copying 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. Upon a set operation (using the UVALUE keyword to WIDGET_TAB or the SET_UVALUE keyword to WIDGET_CONTROL), the variable passed as value becomes undefined. Upon 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 containing 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 callback procedure. This callback procedure can be removed by setting the routine name 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 to the user.

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.

UNAME

Set this keyword to a string, which is 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 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 (which 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.

XOFFSET

The horizontal 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 plain base widget. You should avoid using this style of widget programming.

XSIZE

The width of the widget in units specified by the UNITS keyword (pixels are the default). Most widgets attempt to size themselves to fit the situation. However, if the desired effect is not produced, use this keyword to override it. This keyword is only a hint to the toolkit and may be ignored in some situations.

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 plain base widget. You should avoid using this style of widget programming.

YSIZE

The height of the widget in units specified by the UNITS keyword (pixels are the default). Most widgets attempt to size themselves to fit the situation. However, if the desired effect is not produced, use this keyword to override it. This keyword is only a hint to the toolkit and may be ignored in some situations.

Keywords to WIDGET_CONTROL

A number of keywords to the WIDGET_CONTROL affect the behavior of tab widgets. In addition to those keywords that affect all widgets, the following keywords are particularly useful: BASE_SET_TITLE, SET_TAB_CURRENT, SET_TAB_MULTILINE.

Keywords to WIDGET_INFO

Some keywords to the WIDGET_INFO return information that applies specifically to tab widgets. In addition to those keywords that apply to all widgets, the following keywords are particularly useful: TAB_CURRENT, TAB_MULTILINE, TAB_NUMBER.

Widget Events Returned by Tab Widgets

Tab widgets generate events when a new tab is selected. The event structure returned by the WIDGET_EVENT function is defined by the following statement:

```
{WIDGET_TAB, ID:0L, TOP:0L, HANDLER:0L, TAB:0L}
```

ID is the widget ID of the button generating the event. TOP is the widget ID of the top level widget containing ID. HANDLER contains the widget ID of the widget associated with the handler routine. TAB returns the zero-based index of the tab selected.

Version History

Introduced: 5.6

See Also

“Using Tab Widgets” in Chapter 26 of the *Building IDL Applications* manual

WIDGET_TREE

The WIDGET_TREE function is used to create and populate a tree widget. The tree widget presents a hierarchical view that can be used to organize a wide variety of data structures and information.

The WIDGET_TREE function performs two separate tasks: creating the tree widget and populating the tree widget with *nodes* (branches and leaves).

For a more detailed discussion of the tree widget, along with examples, see "Using Tree Widgets" in Chapter 26 of the *Building IDL Applications* manual.

Syntax

Result = WIDGET_TREE(*Parent* [, /ALIGN_BOTTOM | , /ALIGN_CENTER | , /ALIGN_LEFT | , /ALIGN_RIGHT | , /ALIGN_TOP] [, BITMAP=*array*] [, /CONTEXT_EVENTS] [, EVENT_FUNC=*string*] [, EVENT_PRO=*string*] [, /EXPANDED] [, /FOLDER] [, FUNC_GET_VALUE=*string*] [, GROUP_LEADER=*widget_id*] [, KILL_NOTIFY=*string*] [, /MULTIPLE] [, /NO_COPY] [, NOTIFY_REALIZE=*string*] [, PRO_SET_VALUE=*string*] [, SCR_XSIZE=*width*] [, SCR_YSIZE=*height*] [, /SENSITIVE] [, /TOP] [, UNAME=*string*] [, UNITS={0 | 1 | 2}] [, UVALUE=*value*] [, VALUE=*string*] [, XOFFSET=*value*] [, XSIZE=*value*] [, YOFFSET=*value*] [, YSIZE=*value*])

Return Value

The returned value of this function is the widget ID of the newly-created tree widget.

Arguments

Parent

The widget ID of the parent for the new tree widget. *Parent* can be either a base widget or a tree widget.

- If *Parent* is a base widget, WIDGET_TREE will create a tree widget that contains no other tree widgets. This type of tree widget is referred to as a *root node*.
- If *Parent* is a tree widget, WIDGET_TREE will create a new tree widget (called a *node*) in the specified tree widget.

Note

With the exception of the first tree widget created (the *root node*, whose *Parent* is a base widget), a tree widget (or *node*) must be created with the FOLDER keyword in order to serve as the *Parent* for other tree widgets.

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.

BITMAP

Set this keyword equal to a 16x16x3 array representing an RGB image that will be displayed next to the node in the tree widget.

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 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.

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.

EXPANDED

If the tree node being created is a *folder* (specified by the FOLDER keyword), set this keyword to cause the folder to be initially displayed expanded, showing all of its immediate child entries. By default, folders are initially displayed collapsed.

This keyword is only valid if the *Parent* of the tree widget is another tree widget.

FOLDER

Set this keyword to cause the tree node being created to act as a *folder* (that is, as a branch of the tree rather than a leaf).

Note

With the exception of the root node (the tree widget whose *Parent* widget is a base widget), only tree nodes that have the FOLDER keyword set can act as the parent for other tree widgets.

This keyword is only valid if the *Parent* of the tree widget is another tree widget.

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.

GROUP_LEADER

The widget ID of an existing widget that serves as group leader for the newly-created widget. When a group leader is killed, for any reason, all widgets in the group are also destroyed.

A given widget can be in more than one group. The WIDGET_CONTROL procedure can be used to add additional group associations to a widget. You cannot remove a widget from an existing group.

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. This callback procedure can be removed by setting the routine name to the null string (`''`). Note that the procedure specified is used only if you are not using the XMANAGER procedure to manage your widgets.

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.

If you use the XMANAGER procedure to manage your widgets, the value of this keyword is overwritten. Use the CLEANUP keyword to XMANAGER to specify a procedure to be called when a managed widget dies.

MULTIPLE

Set this keyword to enable multiple selection operations in the tree widget. If enabled, multiple elements in the tree widget can be selected at one time by holding down the **Control** or **Shift** key while clicking the left mouse button.

This keyword is only valid if the *Parent* of the tree widget is a base widget.

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 works well 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 copying 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. Upon a set operation (using the UVALUE keyword to WIDGET_TAB or the SET_UVALUE keyword to WIDGET_CONTROL), the variable passed as value becomes undefined. Upon 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 containing 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. This callback procedure can be removed by setting the routine name 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 to the user.

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.

TOP

Set this keyword to cause the tree node being created to be inserted as the parent node's top entry. By default, new nodes are inserted as the parent node's bottom entry.

This keyword is only valid if the *Parent* of the tree widget is another tree widget.

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 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 (which 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.

VALUE

Set this keyword equal to a string containing the text that will be displayed next to the tree node. If this keyword is not set, the default value `Tree` is used.

This keyword is only valid if the *Parent* of the tree widget is another tree widget.

XOFFSET

The horizontal 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 plain base widget. You should avoid using this style of widget programming.

XSIZE

The width of the widget in units specified by the UNITS keyword (pixels are the default). Most widgets attempt to size themselves to fit the situation. However, if the desired effect is not produced, use this keyword to override it. This keyword is only a hint to the toolkit and may be ignored in some situations.

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 plain base widget. You should avoid using this style of widget programming.

YSIZE

The height of the widget in units specified by the UNITS keyword (pixels are the default). Most widgets attempt to size themselves to fit the situation. However, if the desired effect is not produced, use this keyword to override it. This keyword is only a hint to the toolkit and may be ignored in some situations.

Keywords to WIDGET_CONTROL

A number of keywords to the WIDGET_CONTROL affect the behavior of tree widgets. In addition to those keywords that affect all widgets, the following keywords are particularly useful: SET_TREE_BITMAP, SET_TREE_EXPANDED, SET_TREE_SELECT, SET_TREE_VISIBLE.

Keywords to WIDGET_INFO

Some keywords to the WIDGET_INFO return information that applies specifically to tree widgets. In addition to those keywords that apply to all widgets, the following keywords are particularly useful: TREE_EXPANDED, TREE_SELECT, and TREE_ROOT.

Widget Events Returned by Tree Widgets

Several variations of the tree 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. Programs should always check the type field before referencing fields that are not present in all tree event structures. The different tree widget event structures are described below.

Select (TYPE = 0)

This structure is returned when the currently selected node in the tree widget changes:

```
{WIDGET_TREE_SEL, ID:0L, TOP:0L, HANDLER:0L, TYPE:0, CLICKS:0L}
```

The CLICKS field indicates the number of mouse-button clicks that occurred when the event took place. This field contains 1 (one) when the item is selected, or 2 when the user double-clicks on the item.

Expand (TYPE = 1)

This structure is returned when a folder in the tree widget expands or collapses:

```
{WIDGET_TREE_EXPAND, ID:0L, TOP:0L, HANDLER:0L, TYPE:1, EXPAND:0L}
```

The EXPAND field contains 1 (one) if the folder expanded or 0 (zero) if the folder collapsed.

Context Menu Events

Tree widgets return the following event structure when the user clicks the right mouse button and the tree widget was created with the CONTEXT_EVENTS keyword set:

```
{WIDGET_CONTEXT, ID:0L, TOP:0L, HANDLER:0L, X:0L, Y:0L}
```

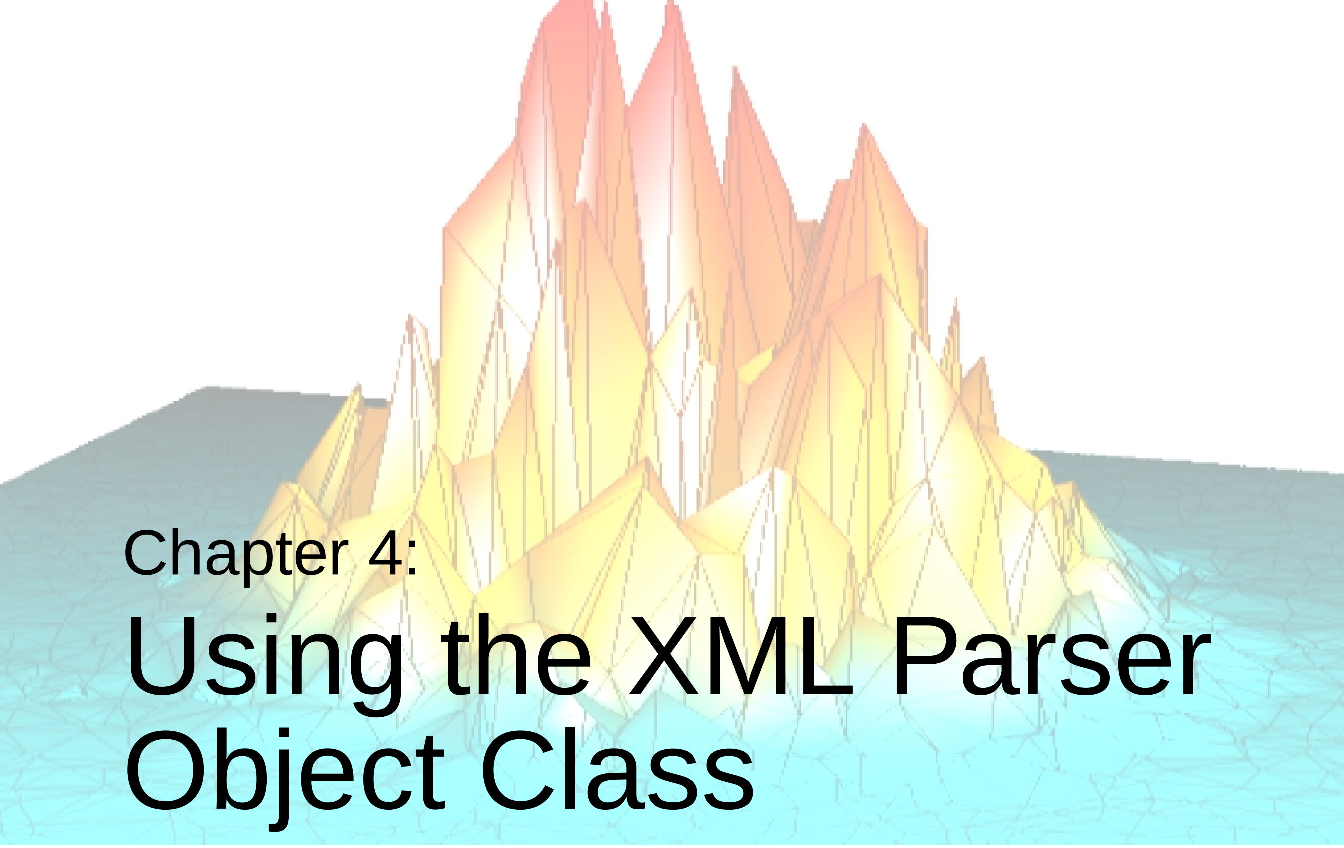
The first three fields are the standard fields found in every widget event. The X and Y fields give the device coordinates at which the event occurred, and are measured from the upper left corner of the tree widget.

Version History

Introduced: 5.6

See Also

"Using Tree Widgets" in Chapter 26 of the *Building IDL Applications* manual



Chapter 4: Using the XML Parser Object Class

The following topics are covered in this chapter:

About XML

XML (eXtensible Markup Language) provides a set of rules for defining semantic tags that can describe virtually any type of data in a simple ASCII text file. Data stored in XML-format files is both human- and machine-readable, and is often relatively easy to interpret either visually or programmatically. The structure of data stored in an XML file is described by either a Document Type Definition (DTD) or an XML schema, which can either be included in the file itself or referenced from an external network location.

It is beyond the scope of this manual to describe XML in detail. Numerous third-party books and electronic resources are available. The following texts may be useful:

- `http://www.w3.org` — information about many web standards, including XML related technologies.
- `http://www.w3schools.com` — tutorials on all manner of XML-related topics.
- `http://www.saxproject.org` — information about the Simple API for XML, the event-based XML parsing technology used by IDL.
- Brownell, David. *SAX2*. O'Reilly & Associates, 2002. ISBN: 0-596-00237-8.
- Harold, Eliotte Rusty. *XML Bible*. IDG Books Worldwide, 1999. ISBN: 0-7645-3236-7

About XML Parsers

There are two basic types of parsers for XML data:

- tree-based parsers
- event-based parsers.

Tree-based Parsers

Tree-based parsers map an XML document into a tree structure in memory, allowing you to select elements by navigating through the tree. This type of parser is generally based on the Document Object Model (DOM) and the tree is often referred to as a DOM tree.

Tree-based parsers are especially useful when the XML data file being parsed is relatively small. Having access to the entire data set at one time can be convenient and makes processing data based on multiple data values stored in the tree easy.

However, if the tree structure is larger than will fit in physical memory or if the data must be converted into a new (local) data structure before use, then tree-based parsers can be slow and cumbersome.

Event-based Parsers

Event-based parsers read the XML document sequentially and report parsing events (such as the start or end of an element) as they occur, without building an internal representation of the data structure. The most common examples of event-based XML parsers use the Simple API for XML (SAX), and are often referred to as a SAX parsers.

Event-based parsers allow the programmer to write *callback routines* that perform an appropriate action in response to an event reported by the parser. Using an event-based parser, you can parse very large data files and create application-specific data structures. The IDLffXMLSAX object class implements an event-based parser based on the SAX version 2 API.

Using the XML Parser

IDL's XML parser object class (IDLffXMLSAX) implements a SAX 2 event-based parser. The object's methods are a set of *callback routines* that are called automatically when the parser encounters different constituents of an XML document. For example, when the parser encounters the beginning of an XML element, it calls the `StartElement` method. When the `StartElement` method returns, the parser continues.

The IDLffXMLSAX object's methods are completely generic. As provided, they do nothing with the items encountered in the XML file. To use the parser object to read data from an XML file, you *must* write a subclass of the IDLffXMLSAX class, overriding the superclass's methods to accomplish your objectives. This requirement that you subclass the object makes the IDLffXMLSAX class unlike any other object class supplied by IDL.

For a detailed discussion of IDL object classes, subclassing, and method overriding, see Chapter 21, "Object Basics" in *Building IDL Applications*. For a description of the parser object class and its methods, see "IDLffXMLSAX object" on page 146.

Subclassing the IDLffXMLSAX Object Class

Writing a subclass of the IDLffXMLSAX object class is similar to writing a subclass of any of IDL's other object classes. The basic steps are:

1. Define a class structure for your subclass, inheriting from the IDLffXMLSAX object class.
2. Write methods to override the IDLffXMLSAX object class methods as necessary.
3. Write additional methods required for your application.
4. Create a class definition routine for your XML parser object.

Let's look at these steps individually:

Define a Class Structure

Every object class has a unique class structure that defines the instance data contained in the object. (See "Class Structures" on page 491 in *Building IDL Applications* for details.) When writing your own parser object (a subclass of the IDLffXMLSAX object), you must first determine what instance data you need your parser object to contain, and define a class structure accordingly.

Note

Your parser object's class structure must inherit from the IDLffXMLSAX class structure. See "Inheritance" on page 493 in *Building IDL Applications* for details.

For example, suppose you want to use your parser to extract an array of data from an XML file. You might choose to define your class structure to include an IDL pointer that will contain the data array. For this case, your class structure definition might look something like

```
void = {myParser, INHERITS IDLffXMLSAX, ptr:PTR_NEW()}
```

Within your subclass's methods, this data structure will always be available via the implicit `self` argument (see "Method Routines" on page 506 in *Building IDL Applications* for details). Setting the value of `self.ptr` within a method routine sets the instance data of the object.

In most cases, your class structure definition will be included in a routine that does *Automatic Structure Definition* (see "Automatic Class Structure Definition" on page 492 in *Building IDL Applications*).

Override Superclass Methods

For your XML parser to do any work, you must override the generic methods of the IDLffXMLSAX object class. Overriding a method is as simple as defining a method routine with the same name as the superclass's method. When your parser encounters an item in the parsed XML file that triggers one of the IDLffXMLSAX methods, it will look first for a method of the same name in the definition of your subclass of the IDLffXMLSAX object class. See "Method Overriding" on page 510 in *Building IDL Applications* for details.

For example, suppose you want your parser to print out the element name of each XML element it encounters to IDL's output. You could override the `StartElement` method of the IDLffXMLSAX class as follows:

```
PRO myParser::StartElement, URI, Local, Name

PRINT, Name

END
```

Note

The new method must take the same parameters as the overridden method.

When your parser encounters the beginning of an XML element, it will look for a method named `StartElement` and call that method with the parameters specified for the IDLffXMLSAX::StartElement method. Since your subclass's StartElement method is found before the superclass's StartElement method, your method is used.

Note

You do not necessarily need to override *all* of the IDLffXMLSAX object methods. Depending on your application, it may be sufficient to override four or five of the superclass's methods. See the parser definitions later in this chapter for examples.

Overriding the IDLffXMLSAX methods is the heart of writing your own XML parser. To write an efficient parser, you will need detailed knowledge of the structure of the XML file you want to parse.

See "Example: Reading Data Into an Array" on page 485 and "Example: Reading Data Into Structures" on page 492 for examples of how to work with parsed XML data and return the data in IDL variables.

Write Additional Methods

Depending on your application, you may need to write additional object methods to work with the instance data retrieved from the parsed XML file. Like the overridden object methods, any new methods you write have access to the object's instance data via the implicit `self` parameter.

Create a Class Definition Routine

If you combine your class definition routine with your class's method routines in a file, you can use IDL's *Automatic Structure Definition* feature to automatically compile the class routines when an instance of your class is created via the OBJ_NEW function. Keep the following in mind when creating the `.pro` file that will contain the definition of your class structure and method routines:

- The routine that creates your class structure should be named with the characters "__define" appended to the end of the class name. For example, if your parser object class is named "myParser" and its class structure is the one described in "Define a Class Structure" on page 480, the routine definition would be:

  ```
  PRO myParser__define

  void = {myParser, INHERITS IDLffXMLSAX, ptr:PTR_NEW()}

  END
  ```

- The .pro file should be named after the class structure definition routine. In this case, the name would be myParser__define.pro.
- The class structure definition routine should be the last routine in the .pro file.

Using Your Parser

Once you have written the class definition routine for your parser, you are ready to parse an XML file. The process is straightforward:

1. Create an instance of your parser object.
2. Call the ParseFile method on your object instance with the name of an XML file as the parameter.

For example, if your parser object is named myParser and the object class definition file is named myParser__define.pro, you could use the following IDL statements:

```
xmlFile = OBJ_NEW('myParser')
xmlFile -> ParseFile, 'data.xml'
```

The first statement creates a new XML parser based on your class definition and places a reference to the parser object in the variable xmlFile. The second statement calls the ParseFile method on that object with the filename data.xml.

What happens next depends on your application. If your object definition stores values from the parsed file in the object's instance data, you will need some way to retrieve the values into IDL variables that are accessible outside the object. See "Example: Reading Data Into an Array" on page 485 and "Example: Reading Data Into Structures" on page 492 for examples that return data variables that are accessible to other routines.

Validation

An XML document is said to be *valid* if it adheres to a set of constraints set forth in either a Document Type Definition (DTD) or an XML schema. Both DTDs and schemas define which elements can be included in an XML file and what values those elements can assume. XML schemas are a newer technology that is designed to replace and be more robust than DTDs. In working with existing XML files, you are likely to encounter both types of validation mechanisms.

Ensuring that a file contains valid XML helps in writing an efficient parsing mechanism. For example, if your validation method specifies that element B can only occur inside element A, and the XML document you are parsing is known to be valid, then your parser can assume that if it encounters element B it is inside element A.

The IDLffXMLSAX parser object can check an XML document using either validation mechanism, depending on whether a DTD or a schema definition is present. By default, if either is present, the parser will attempt to validate the XML document. See SCHEMA_CHECKING (Get, Set) and VALIDATION_MODE (Get, Set) under “IDLffXMLSAX::Init” on page 167 for details.

Example: Reading Data Into an Array

This example subclasses the IDLffXMLSAX parser object class to create an object class named `xml_to_array`. The `xml_to_array` object class is designed to read numerical values from an XML file with the following structure:

```
<array>
  <number>0</number>
  <number>1</number>
  ...
</array>
```

and place those values into an IDL array variable.

Note

This example is a very simple example. It is designed to illustrate how an event-based XML parser is constructed using the IDLffXMLSAX object class. An application that reads real data from an XML file will most likely be quite a bit more complicated.

Creating the xml_to_array Object Class

In order to read the XML file and return an array variable, we will need to create an object class definition that inherits from the IDLffXMLSAX object class, and override the following superclass methods: `Init`, `Cleanup`, `StartDocument`, `Characters`, `StartElement`, and `EndElement`. Since this example does not retrieve data using any of the other IDLffXMLSAX methods, we do not need to override those methods. In addition, we will create a new method that allows us to retrieve the array data from the object instance data.

Note

This example is included in the file `xml_to_array__define.pro` in the `examples/data_access` subdirectory of the IDL distribution.

Object Class Definition

The following routine is the definition of the `xml_to_array` object class:

```
PRO xml_to_array__define

void = {xml_to_array, $
   INHERITS IDLffXMLSAX, $
   charBuffer:'', $
   pArray:PTR_NEW()}
END
```

The following items should be considered when defining this class structure:

- The structure definition uses the INHERITS keyword to inherit the object class structure and methods of the IDLffXMLSAX object.
- The `charBuffer` structure field is set equal to an empty string.
- The `pArray` structure field is set equal to an IDL pointer. We will use this pointer to store the numerical array data we retrieve.
- The routine name is created by adding the string "`__define`" (note the *two* underscore characters) to the class name.

Why do we store the array data in a pointer variable? Because the fields of a named structure (`xml_to_array`, in this case) must always contain the same type of data as when that structure was defined. Since we want to be able to add values to the data array as we parse the XML file, we will need to extend the array with each new value. If we began by defining the size of the array in the structure variable, we would not be able to extend the array. By holding the data array in a pointer, we can extend the array without changing the format of the `xml_to_array` object class structure.

Note

Although we describe this routine first here, the `xml_to_array__define` routine must be the *last* routine in the `xml_to_array__define.pro` file.

Init Method

The `Init` method is called when the an `xml_to_array` parser object is created by a call to OBJ_NEW. The following routine is the definition of the `Init` method:

```
FUNCTION xml_to_array::Init
  self.pArray = PTR_NEW(/ALLOCATE_HEAP)
  RETURN, self -> IDLffxmlsax::Init()
END
```

We do two things in this method:

- We initialize the pointer in the `pArray` field of the class structure variable.

  Note

  Within a method, we can refer to the class structure variable with the implicit parameter `self`. Remember that `self` is actually a reference to the `xml_to_array` object instance.

- The return value from this function is the return value of the superclass's `Init` method, called on the `self` object reference.

Note

The initialization task (setting the value of the `pArray` field) is performed before calling the superclass's `Init` method.

See "IDLffXMLSAX::Init" on page 167 for details on the method we are overriding.

Cleanup Method

The `Cleanup` method is called when the `xml_to_array` parser object is destroyed by a call to OBJ_DESTROY. The following routine is the definition of the `Cleanup` method:

```
PRO xml_to_array::Cleanup

IF (PTR_VALID(self.pArray)) THEN PTR_FREE, self.pArray

END
```

All we do in the `Cleanup` method is to release the `pArray` pointer, if it exists.

See "IDLffXMLSAX::Cleanup" on page 152 for details on the method we are overriding.

Characters Method

The `Characters` method is called when the `xml_to_array` parser encounters character data inside an element. The following routine is the definition of the `Characters` method:

```
PRO xml_to_array::characters, data

self.charBuffer = self.charBuffer + data

END
```

As it parses the character data in an element, the parser will read characters until it reaches the end of the text section. Here, we simply add the current characters to the `charBuffer` field of the object's instance data structure.

See "IDLffXMLSAX::Characters" on page 151 for details on the method we are overriding.

StartDocument Method

The `StartDocument` method is called when the `xml_to_array` parser encounters the beginning of the XML document. The following routine is the definition of the `StartDocument` method:

```
PRO xml_to_array::StartDocument

IF (N_ELEMENTS(*self.pArray) GT 0) THEN $
   void = TEMPORARY(*self.pArray)

END
```

Here, we check to see if the array pointed at by the `pArray` pointer contains any data. Since we are just beginning to parse the XML document at this point, it should not contain any data. If data is present, we reinitialize the array using the TEMPORARY function.

Note

Since `pArray` is a pointer, we must use dereferencing syntax to refer to the array.

See "IDLffXMLSAX::StartDocument" on page 176 for details on the method we are overriding.

StartElement Method

The `StartElement` method is called when the `xml_to_array` parser encounters the beginning of an XML element. The following routine is the definition of the `StartElement` method:

```
PRO xml_to_array::startElement, URI, local, strName, attr, value

CASE strName OF
   "array": BEGIN
      IF (N_ELEMENTS(*self.pArray) GT 0) THEN $
      void = TEMPORARY(*self.pArray);; clear out memory
   END
   "number" : BEGIN
      self.charBuffer = ''
   END
```

```
ENDCASE

END
```

Here, we first check the name of the element we have encountered, and use a CASE statement to branch based on the element name:

- If the element is an `<array>` element, we check to see if the array pointed at by the `pArray` pointer is empty. Since we are just beginning to read the array data at this point, there should be no data. If data already exists, we reinitialize the array using the TEMPORARY function.
- If the element is a `<number>` element, we reinitialize the `charBuffer` field. Since we are just beginning to read the number data, nothing should be in the buffer.

See "IDLffXMLSAX::StartElement" on page 178 for details on the method we are overriding.

EndElement Method

The `EndElement` method is called when the `xml_to_array` parser encounters the end of an XML element. The following routine is the definition of the `EndElement` method:

```
PRO xml_to_array::EndElement, URI, Local, strName

CASE strName OF
   "array":
   "number": BEGIN
      idata = FIX(self.charBuffer);
      IF (N_ELEMENTS(*self.pArray) EQ 0) THEN $
         *self.pArray = iData $
      ELSE $
         *self.pArray = [*self.pArray,iData]
   END
ENDCASE

END
```

As with the `StartElement` method, we first check the name of the element we have encountered, and use a CASE statement to branch based on the element name:

- If the element is an `<array>` element, we do nothing.
- If the element is a `<number>` element, we must get the data stored in the `charBuffer` field of the instance data structure and place it in the array:
  - First, we convert the string data in the `charBuffer` into an IDL integer.

- Next, we check to see if the array pointed at by `pArray` is empty. If it is empty, we simply set the array equal to the data value we retrieved from the `charBuffer`.
- If the array pointed at by `pArray` is not empty, we redefine the array to include the new data retrieved from the `charBuffer`.

See "IDLffXMLSAX::EndElement" on page 158 for details on the method we are overriding.

Note

In both the `StartElement` and `EndElement` methods, we rely on the validity of the XML data file. Our CASE statements only need to handle the element types described in the XML file's DTD or schema (in this case, the only elements are `<array>` and `<number>`). We do not need an ELSE clause in the CASE statement. If an unknown element is found in the XML file, the parser will report a validation error.

GetArray Method

The `GetArray` method allows us to retrieve the array data stored in the `pArray` pointer variable. The following routine is the definition of the `GetArray` method:

```
FUNCTION xml_to_array::GetArray

IF (N_ELEMENTS(*self.pArray) GT 0) THEN $
   RETURN, *self.pArray $
ELSE RETURN , -1

END
```

Here, we check to see whether the array pointed at by `pArray` contains any data. If it does contain data, we return the array. If the array contains no data, we return the value `-1`.

Using the xml_to_array Parser

To see the `xml_to_array` parser in action, you can parse the file `num_array.xml`, found in the `examples/data` subdirectory of the IDL distribution. This `num_array.xml` file contains the fragment of XML like the one shown in the beginning of this section, and includes 20 extra `<number>` elements. The `num_array.xml` file also includes a DTD describing the structure of the file.

Enter the following statements at the IDL command line:

```
xmlObj = OBJ_NEW('xml_to_array')
xmlFile = FILEPATH('num_array.xml', $
   SUBDIRECTORY = ['examples', 'data'])
xmlObj -> ParseFile, xmlFile
myArray = xmlObj -> GetArray()
OBJ_DESTROY, xmlObj
HELP, myArray
PRINT, myArray
```

IDL prints:

```
MYARRAY         INT       = Array[20]
 0   1   2   3   4   5   6   7   8   9   10   11
12  13  14  15  16  17  18  19
```

Example: Reading Data Into Structures

This example subclasses the IDLffXMLSAX parser object class to create an object class named `xml_to_struct`. The `xml_to_struct` object class is designed to read data from an XML file with the following structure:

```
<Solar_System>
   <Planet NAME='Mercury'>
      <Orbit UNITS='kilometers' TYPE='ulong64'>579100000</Orbit>
      <Period UNITS='days' TYPE='float'>87.97</Period>
      <Satellites TYPE='int'>0</Satellites>
   </Planet>
  ...
</Solar_System>
```

and place those values into an IDL array containing one structure variable for each `<Planet>` element. We use a structure variable for each `<Planet>` element so we can capture data of several data types in a single place.

Note

While this example is more complicated than the previous example, it is still rather simple. It is designed to illustrate a method whereby more complex XML data structures can be represented in IDL.

Creating the xml_to_struct Object Class

To read the XML file and return a structure variable, we will need to create an object class definition that inherits from the IDLffXMLSAX object class, and override the following superclass methods: `Init`, `Characters`, `StartElement`, and `EndElement`. Since this example does not retrieve data using any of the other IDLffXMLSAX methods, we do not need to override those methods. In addition, we will create a new method that allows us to retrieve the structure data from the object instance data.

Notice that the elements of the XML data file include *attributes*. While we will retrieve and use some of the attribute data from the file, we will ignore some of it.

Note

When parsing an XML data file, you can pick and choose the data you wish to pull into IDL. This ability to selectively retrieve data from the XML file is one of the great advantages of an event-based parser over a tree-based parser.

Note

This example is included in the file `xml_to_struct__define.pro` in the `examples/data_access` subdirectory of the IDL distribution.

Object Class Definition

The following routine is the definition of the `xml_to_struct` object class:

```
PRO xml_to_struct__define

void = {PLANET, NAME: "", Orbit: 0ull, period:0.0, Moons:0}
void = {xml_to_struct, $
   INHERITS IDLffXMLSAX, $
   CharBuffer:"", $
   planetNum:0, $
   currentPlanet:{PLANET}, $
   Planets : MAKE_ARRAY(9, VALUE = {PLANET})}

END
```

The following items should be considered when defining this class structure:

- Before creating the object class structure, we define a structure named PLANET. We will use the PLANET structure to store data from the `<Planet>` elements of the XML file.
- The object class structure definition uses the `INHERITS` keyword to inherit the object class structure and methods of the IDLffXMLSAX object.
- The `charBuffer` structure field is set equal to a string value. We will use this field to accumulate character data stored in XML elements.
- The `planetNum` structure field is set equal to an integer value. We will use this field to keep track of which array element we are currently populating.
- The `currentPlanet` structure field is set equal to a PLANET structure.
- The `Planets` structure field is set equal to a nine-element array of PLANET structures.
- The routine name is created by adding the string "`__define`" (note the *two* underscore characters) to the class name.

We have explicitly defined our `Planets` structure field as a nine-element array of PLANET structures, which we can do because we know exactly how many `<Planet>` elements will be read from our XML file. Specifying the exact size of the data array in the class structure definition is very efficient (since we create the array only once) and eliminates the need to free the pointer in the `Cleanup` method. However, it has the following consequences:

- We must explicitly keep track of the index of the array element we are populating, and increment it after we have finished with a given element (see the `EndElement` method below).
- We must know in advance how many elements the array will hold. If the size of the final array is unknown, it is more efficient to use a pointer to an array, as we did in the previous example, and allow the array to grow as elements are added. See "Building Complex Data Structures" on page 499 for additional discussion of ways to configure the instance data structure.

Note

Although we describe this routine here first, the `xml_to_struct__define` routine must be the last routine in the `xml_to_struct__define.pro` file.

Init Method

The `Init` method is called when the an `xml_to_struct` parser object is created by a call to OBJ_NEW. The following routine is the definition of the `Init` method:

```
FUNCTION xml_to_struct::Init

self.planetNum = 0
RETURN, self -> IDLffXMLSAX::Init()

END
```

We do two things in this method:

- We initialize the `planetNum` field with the value of zero. We will increment this value as we populate the `Planets` array.

  Note

  Within a method, we can refer to the class structure variable with the implicit parameter `self`. Remember `self` is actually a reference to the `xml_to_struct` object instance.

- The return value from this function is the return value of the superclass's `Init` method, called on the `self` object reference.

Note

We perform our own initialization task (setting the value of the `planetNum` field) before calling the superclass's `Init` method.

See "IDLffXMLSAX::Init" on page 167 for details on the method we are overriding.

Characters Method

The `Characters` method is called when the `xml_to_struct` parser encounters character data inside an element. The following routine is the definition of the `Characters` method:

```
PRO xml_to_struct::characters, data

self.charBuffer = self.charBuffer + data

END
```

As it parses the character data in an element, the parser will read characters until it reaches the end of the text section. Here, we simply add the current characters to the `charBuffer` field of the object's instance data structure.

See "IDLffXMLSAX::Characters" on page 151 for details on the method we are overriding.

StartElement Method

The `StartElement` method is called when the `xml_to_struct` parser encounters the beginning of an XML element. The following routine is the definition of the `StartElement` method:

```
PRO xml_to_struct::startElement, URI, local, strName, attrName, attrValue

CASE strName OF
   "Solar_System":   ; Do nothing
   "Planet" : BEGIN
      self.currentPlanet = {PLANET, "", 0ull, 0.0, 0}
      self.currentPlanet.Name = attrValue[0]
   END
   "Orbit" : self.charBuffer = ''
   "Period" : self.charBuffer = ''
   "Moons" : self.charBuffer = ''
ENDCASE

END
```

Here, we first check the name of the element we have encountered, and use a CASE statement to branch based on the element name:

- If the element is a `<Solar_System>` element, we do nothing.
- If the element is a `<Planet>` element, we do the following things:
  - Set the value of the `currentPlanet` field of the `self` instance data structure equal to a PLANET structure, setting the values of the structure fields to zero values.
  - Set the value of the `Name` field of the PLANET structure held in the `currentPlanet` field equal to the value of the `Name` attribute of the element. This field contains the name of the planet whose data we are reading.
- If the element is an `<Orbit>`, `<Period>`, or `<Moons>` element, we reinitialize the value of the `charBuffer` field of the `self` instance data structure.

See "IDLffXMLSAX::StartElement" on page 178 for details on the method we are overriding.

EndElement Method

The `EndElement` method is called when the `xml_to_struct` parser encounters the end of an XML element. The following routine is the definition of the `EndElement` method:

```
PRO xml_to_struct::EndElement, URI, Local, strName

CASE strName of
   "Solar_System":
   "Planet": BEGIN
      self.Planets[self.planetNum] = self.currentPlanet
      self.planetNum = self.planetNum + 1
   END
   "Orbit" : self.currentPlanet.Orbit = self.charBuffer
   "Period" : self.currentPlanet.Period = self.charBuffer
   "Moons" : self.currentPlanet.Moons= self.charBuffer
ENDCASE

END
```

As with the `StartElement` method, we first check the name of the element we have encountered, and use a CASE statement to branch based on the element name:

- If the element is a `<Solar_System>` element, we do nothing.
- If the element is a `<Planet>` element, we set the element of the `Planets` array specified by `planetNum` equal to the PLANET structure contained in `currentPlanet`. Then, we increment the `planetNum` counter.
- If the element is an `<Orbit>`, `<Period>`, or `<Satellites>` element, we place the value in the `charBuffer` field into the appropriate field within the PLANET structure contained in `currentPlanet`.

See "IDLffXMLSAX::EndElement" on page 158 for details on the method we are overriding.

Note

In both the `StartElement` and `EndElement` methods, we rely on the validity of the XML data file. Our CASE statements only need to handle the element types described in the XML file's DTD or schema. We do not need an ELSE clause in the CASE statement. If an unknown element is found in the XML file, the parser will report a validation error.

GetArray Method

The `GetArray` method allows us to retrieve the array of structures stored in the `Planets` variable. The following routine is the definition of the `GetArray` method:

```
FUNCTION xml_to_struct::GetArray

IF (self.planetNum EQ 0) THEN $
   RETURN, -1 $
ELSE RETURN, self.Planets[0:self.planetNum-1]

END
```

Here, we check to see whether the `planetNum` counter has been incremented. If it has been incremented, we return as the number of array elements specified by the counter. If the counter has not been incremented (indicating that no data has been stored in the array), we return the value `-1`.

Using the xml_to_struct Parser

To see the `xml_to_struct` parser in action, you can parse the file `planets.xml`, found in the `examples/data` subdirectory of the IDL distribution. The `planets.xml` file contains the fragment of XML like the one shown at the beginning of this section, and includes a `<Planet>` element for each planet in the solar system. The `planets.xml` file also includes a DTD describing the structure of the file.

Enter the following statements at the IDL command line:

```
xmlObj = OBJ_NEW('xml_to_struct')
xmlFile = FILEPATH('planets.xml', $
   SUBDIRECTORY = ['examples', 'data'])
xmlObj -> ParseFile, xmlFile
planets = xmlObj -> GetArray()
OBJ_DESTROY, xmlObj
```

The variable `planets` now holds an array of PLANET structures, one for each planet. To print the number of moons for each planet, you could use the following IDL statement:

```
FOR i = 0, (N_ELEMENTS(planets.Name) - 1) DO $
   PRINT, planets[i].Name, planets[i].Moons, $
   FORMAT = '(A7, " has ", I2, " moons")'
```

IDL prints:

```
Mercury has  0 moons
Venus   has  0 moons
Earth   has  1 moons
Mars    has  2 moons
Jupiter has 16 moons
Saturn  has 18 moons
Uranus  has 21 moons
Neptune has  8 moons
Pluto   has  1 moons
```

To view all the information about the planet Mars, you could use the following IDL statement:

```
HELP, planets[3], /STRUCTURE
```

IDL prints:

```
** Structure PLANET, 4 tags, length=32, data length=26:
   NAME       STRING      'Mars'
   ORBIT      ULONG64     227940000
   PERIOD     FLOAT       686.980
   MOONS      INT         2
```

Building Complex Data Structures

Few limitations exist regarding the complexity of the data structures that can be represented in an XML data file. Writing a parser to read data from such complex structures into IDL can be a challenge. If you are designing a parser to read a very complex or deeply nested XML file, keep the following concepts in mind.

Use Dynamically Sized Arrays if Necessary

If you don’t know the final size of your data array, or if the size of the array will change, store the data array in an IDL pointer in the instance data structure. This technique allows you to change the size of the data array without changing the definition of the instance data structure. The downside of extending the data array in this manner is performance. Each time the array is extended, IDL must hold two copies of the entire array in memory. If the array becomes large, this duplication can cause performance problems.

In “Example: Reading Data Into an Array” on page 485, we extended our data array as we added each element despite the fact that we knew the number of data elements. We used a pointer to illustrate the technique, and to make it clear that if you use pointers to store your instance data, you must free the pointers in your subclass’s `Cleanup` method.

Use Fixed-Size Arrays When Possible

If you will be building a large data array, and you know in advance how many elements it will contain, create the array when defining the class data structure and use array indexing to place data in the appropriate elements. Using a fixed-size array eliminates the need to copy the full array each time it is extended, and can lead to noticeable performance improvements when large arrays are involved.

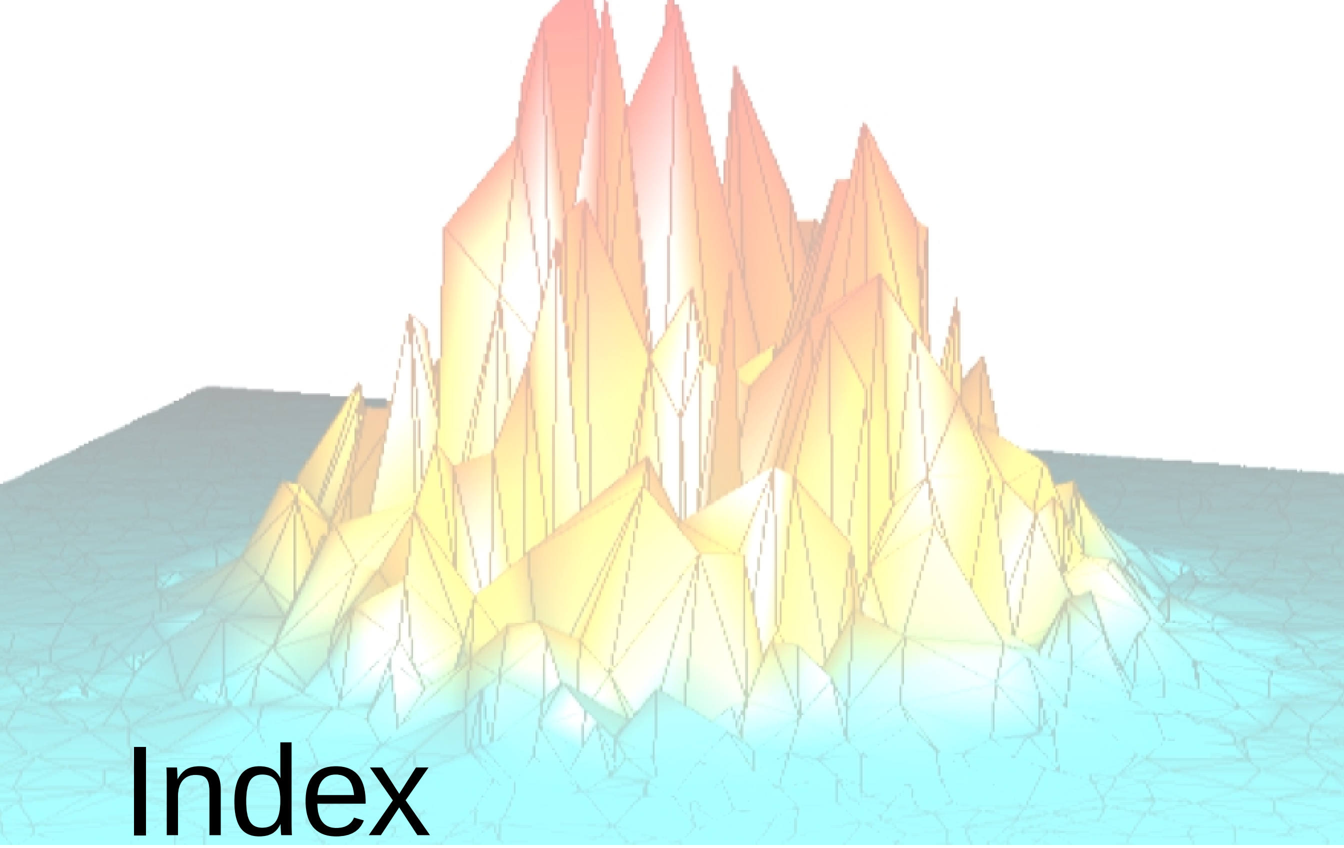
In “Example: Reading Data Into Structures” on page 492, we illustrated the technique of using a pre-defined array to store our instance data.

Using Nested Structures

If your data structure is complex, you may be inclined to represent your data as a set of nested IDL structure variables. While nesting structure variables can help you create a data structure that emulates the structure of your XML file, deeply nested structures can make your code more difficult to create and maintain. Consider storing data in several arrays of structures rather than a single, deeply-nested structure.

If you have a good reason to create nested structures, and also need to extend them dynamically, you should use the CREATE_STRUCT function.

The same caveats apply to extending a structure with CREATE_STRUCT as apply to extending an array. With large datasets, the process of duplicating the structures may cause performance problems.



Index