



# Building IDL Applications



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

## Restricted Rights Notice

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

## Limitation of Warranty

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

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

## Permission to Reproduce this Manual

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

## Acknowledgments

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

Numerical Recipes<sup>™</sup> is a trademark of Numerical Recipes Software. Numerical Recipes routines are used by permission.

GRG2<sup>™</sup> is a trademark of Windward Technologies, Inc. The GRG2 software for nonlinear optimization is used by permission.

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

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

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



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



# Contents

<b>Chapter 1:</b>	
<b>Overview</b> .....	<b>13</b>
What is an IDL Application? .....	14
About Building Applications in IDL .....	15

## **Part I: How to Build Applications in IDL**

<b>Chapter 2:</b>	
<b>Creating IDL Projects</b> .....	<b>19</b>
Overview .....	20
Where to Store the Files for a Project .....	22
Creating a Project .....	23
Opening, Closing, and Saving Projects .....	25
Adding, Moving, and Removing Files .....	26

Working with Files in a Project .....	29
Setting the Options for a Project .....	33
Selecting the Build Order .....	36
Compiling an Application from a Project .....	38
Building a Project .....	39
Running an Application from a Project .....	41
Exporting a Project .....	42
About Developer's Kit Licenses .....	44

### **Chapter 3:**

## **Distributing IDL Applications ..... 45**

Overview .....	46
Creating Your Product Distribution Through Your IDL Project .....	51
Customizing A Windows Distribution .....	55
Customizing a Macintosh Distribution .....	61
Customizing A UNIX Distribution .....	64
For Applications That Use IDL DataMiner .....	65
For Applications That Use ActiveX .....	68
Using the make_rt Script .....	69
Adding IDL Files to the Distribution .....	73
Replacing the Licensing Dialog Image .....	76

## **Part II: Components of IDL**

### **Chapter 4:**

## **The Structure of the IDL Language ..... 81**

Data Types .....	82
Numeric Constants .....	84
String Constants .....	86
Type Conversion Functions .....	87
Arrays .....	90
Structures .....	93
Variables .....	96

System Variables .....	99
<b>Chapter 5:</b>	
<b>Constants .....</b>	<b>101</b>
Data Types .....	102
Constants .....	104
Type Conversion Functions .....	110
<b>Chapter 6:</b>	
<b>Expressions and Operators .....</b>	<b>113</b>
Overview .....	114
Operator Precedence .....	115
IDL Operators .....	117
Type and Structure of Expressions .....	127
<b>Chapter 7:</b>	
<b>Structures .....</b>	<b>131</b>
Overview .....	132
Creating and Defining Structures .....	133
Structure References .....	136
Using HELP with Structures .....	139
Parameter Passing with Structures .....	140
Arrays of Structures .....	143
Structure Input/Output .....	145
Advanced Structure Usage .....	147
Automatic Structure Definition .....	149
Relaxed Structure Assignment .....	151
<b>Chapter 8:</b>	
<b>Array Subscripts .....</b>	<b>155</b>
Overview .....	156
Array Subscript Syntax: [ ] vs. ( ) .....	157
Subscript Examples .....	158
Subscript Ranges .....	161

Structure of Subarrays .....	163
Array Subscripts .....	165
Combining Array Subscripts with Others .....	167
Storing Elements with Array Subscripts .....	169
<b>Chapter 9:</b>	
<b>Strings .....</b>	<b>171</b>
Overview .....	172
String Operations .....	173
Non-string and Non-scalar Arguments .....	174
String Concatenation .....	175
Using STRING to Format Data .....	176
Byte Arguments and Strings .....	177
Case Folding .....	179
Whitespace .....	180
Finding the Length of a String .....	182
Substrings .....	183
Splitting and Joining Strings .....	186
Comparing Strings .....	187
Learning About Regular Expressions .....	191
<b>Chapter 10:</b>	
<b>Statements .....</b>	<b>195</b>
Overview .....	196
Components of Statements .....	197
The Assignment Statement .....	198
Blocks .....	204
CASE Statement .....	206
Common Blocks .....	208
FOR Statement .....	211
Function Definition Statement .....	216
GOTO Statement .....	219
IF Statement .....	220

Procedure Call Statement .....	222
Procedure Definition Statement .....	225
REPEAT Statement .....	226
WHILE Statement .....	227

## **Chapter 11:**

### **Pointers ..... 229**

Overview .....	230
Heap Variables .....	231
Creating Heap Variables .....	233
Saving and Restoring Heap Variables .....	234
Pointer Heap Variables .....	235
IDL Pointers .....	236
Operations on Pointers .....	239
Dangling References .....	243
Heap Variable Leakage .....	244
Pointer Validity .....	246
Freeing Pointers .....	247
Pointer Examples .....	248

## **Chapter 12:**

### **Object Basics ..... 255**

Object-Oriented Programming .....	256
IDL Object Overview .....	257
Class Structures .....	259
Inheritance .....	261
Object Heap Variables .....	263
Null Objects .....	265
The Object Lifecycle .....	266
Operations on Objects .....	269
Obtaining Information about Objects .....	271
Method Routines .....	273

Method Overriding .....	277
Object Examples .....	280

## Part III: Programming in IDL

### **Chapter 13: Defining Procedures and Functions ..... 283**

Overview .....	284
Procedure & Function Definitions .....	285
Parameters .....	286
Using Keyword Parameters .....	289
Keyword Inheritance .....	291
Entering Procedure Definitions .....	296
Parameter Passing Mechanism .....	298
Calling Mechanism .....	300
Setting Compilation Options .....	302

### **Chapter 14: Programming in IDL ..... 305**

Overview of Programming in IDL .....	306
Informational Routines .....	307
Program Control Routines .....	312
Expression Evaluation Order .....	314
Avoid IF Statements .....	315
Use Vector and Array Operations .....	317
IDL System Functions and Procedures .....	319
Use Constants of the Correct Type .....	320
Eliminate Invariant Expressions .....	321
Virtual Memory .....	322
IDL Implementation .....	328

### **Chapter 15: Controlling Errors ..... 329**

Overview .....	330
----------------	-----

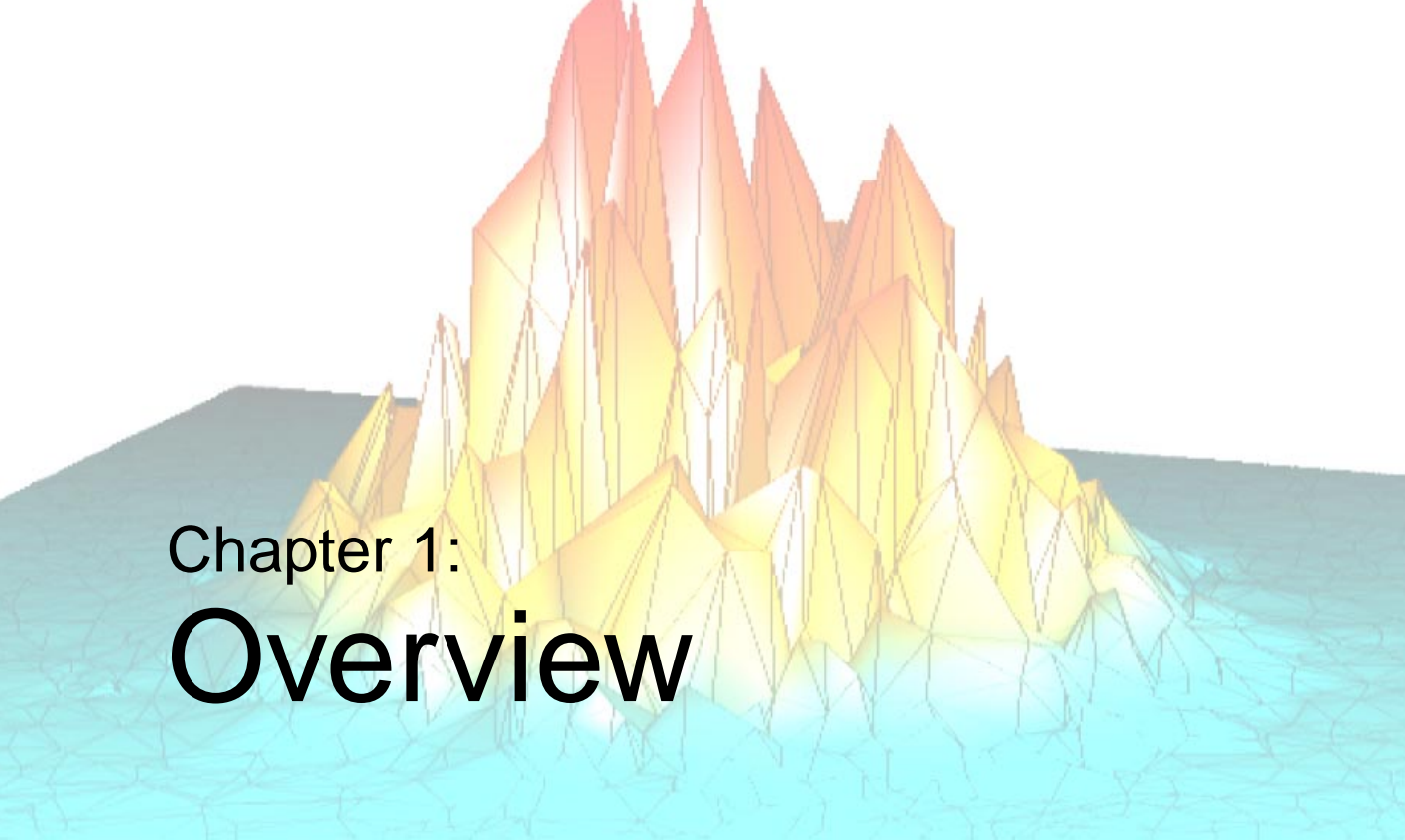


Default Error-Handling Mechanism .....	331
Disappearing Variables .....	332
Controlling Errors Using CATCH .....	333
Controlling Errors Using ON_ERROR .....	337
Controlling Input/Output Errors .....	338
Error Signaling .....	340
Obtaining Traceback Information .....	342
Error Handling .....	343
Math Errors .....	345
<b>Chapter 16:</b>	
<b>Files and Input/Output .....</b>	<b>351</b>
Overview .....	352
File I/O in IDL .....	353
Unformatted Input/Output .....	355
Formatted Input/Output .....	356
Opening Files .....	358
Closing Files .....	359
Logical Unit Numbers (LUNs) .....	360
Reading and Writing Very Large Files .....	363
Using Free Format Input/Output .....	365
Using Explicitly Formatted Input/Output .....	370
Format Codes .....	375
Using Unformatted Input/Output .....	394
Portable Unformatted Input/Output .....	401
Associated Input/Output .....	406
File Manipulation Operations .....	411
UNIX-Specific Information .....	419
VMS-Specific Information .....	422
Windows-Specific Information .....	432
Macintosh-Specific Information .....	433
Scientific Data Formats .....	434

Support for Standard Image File Formats .....	435
<b>Chapter 17:</b>	
<b>Using the IDL GUIBuilder .....</b>	<b>437</b>
Overview .....	438
Starting the IDL GUIBuilder .....	440
Creating an Example Application .....	442
IDL GUIBuilder Tools .....	453
Widget Operations .....	468
Generating Files .....	471
IDL GUIBuilder Examples .....	473
Widget Properties .....	489
Common Widget Properties .....	490
Base Widget Properties .....	496
Button Widget Properties .....	507
Text Widget Properties .....	511
Label Widget Properties .....	516
Slider Widget Properties .....	518
Droplist Widget Properties .....	521
Listbox Widget Properties .....	523
Draw Widget Properties .....	526
Table Widget Properties .....	532
<b>Chapter 18:</b>	
<b>Widgets .....</b>	<b>541</b>
Overview .....	542
Widget Types .....	544
Manipulating Widgets .....	549
Examples of Widget Programming .....	550
The Widget Application Model .....	551
Creating Widget Applications .....	554
Widget Example 1 .....	557
Widget Values .....	559

Widget User Values .....	562
Widget Events .....	563
Widget Example 2 .....	569
Using Draw Widgets .....	571
Creating Menus .....	573
Controlling Widgets .....	578
Widget Example 3 .....	581
Widget Sizing .....	583
Event Processing And Callbacks .....	589
Managing Widget Application State .....	592
Compound Widgets .....	594
Tips on Creating Widget Applications .....	596
Compound Widget Example .....	598
<b>Chapter 19:</b>	
<b>Debugging an IDL Program .....</b>	<b>607</b>
Overview .....	608
Debugging Commands .....	609
The IDL Code Profiler .....	614
The Variable Watch Window .....	619
<b>Chapter 20:</b>	
<b>Building Cross-Platform Applications .....</b>	<b>623</b>
Overview .....	624
Which Operating System is Running? .....	625
File and Path Specifications .....	626
Environment Variables .....	629
Files and I/O .....	630
Math Exceptions .....	633
Operating System Access .....	634
Display Characteristics and Palettes .....	635
Fonts .....	636
Printing .....	637

SAVE and RESTORE .....	638
Widgets .....	639
Using External Code .....	642
IDL DataMiner Issues .....	643
<b>Chapter 21:</b>	
<b>Extending the IDL Online Help System .....</b>	<b>645</b>
Overview .....	646
Creating Hypertext Files for Use with IDL's Hypertext Help Viewer .....	647
<b>Appendix A:</b>	
<b>VMS Floating-Point Arithmetic in IDL .....</b>	<b>649</b>
Overview .....	650
VAX Floating-Point Format Background .....	651
Transition Issues .....	653
A Warning About Floating-Point Conversions in IDL .....	655
A Strategy for Converting VMS Programs .....	656
Using CALL_EXTERNAL .....	658
A Note on the VMS G Float Format .....	660
<b>Index .....</b>	<b>661</b>



# Chapter 1: Overview

This chapter includes information about the following topics:

---

What is an IDL Application? . . . . .	14	About Building Applications in IDL . . . . .	15
---------------------------------------	----	--	----

# What is an IDL Application?

We use the term “IDL Application” very broadly; any program written in the IDL language is, in our view, an IDL application. IDL Applications range from the very simple (a MAIN program entered at the IDL command prompt, for example) to the very complex (large programs with full-blown graphical user interfaces, such as ENVI). Whether you are writing a small program to analyze a single data set or a large-scale application for commercial distribution, it is useful to understand the programming concepts used by the IDL language.

## Can I Distribute My Application?

You can freely distribute IDL source code for your IDL applications to colleagues and others who use IDL. (If you intend to distribute your applications, it is a good idea to avoid any code that depends on the qualities of a specific platform. See “![VERSION](#)” in Appendix D of the *IDL Reference Guide* and “[Creating Widget Applications](#)” on page 554 for some hints on writing platform-independent code.) Of course, IDL applications can only be run from within the IDL environment, so anyone who wishes to run your IDL application must have access to an IDL license.

If you would like to distribute your IDL application to people who do not have access to an IDL license, you may wish to consider a *runtime IDL* licensing agreement. Runtime IDL licenses allow you to distribute a special version of IDL along with your application. Contact your distributor or Research Systems sales representative for information about runtime licensing.

# About Building Applications in IDL

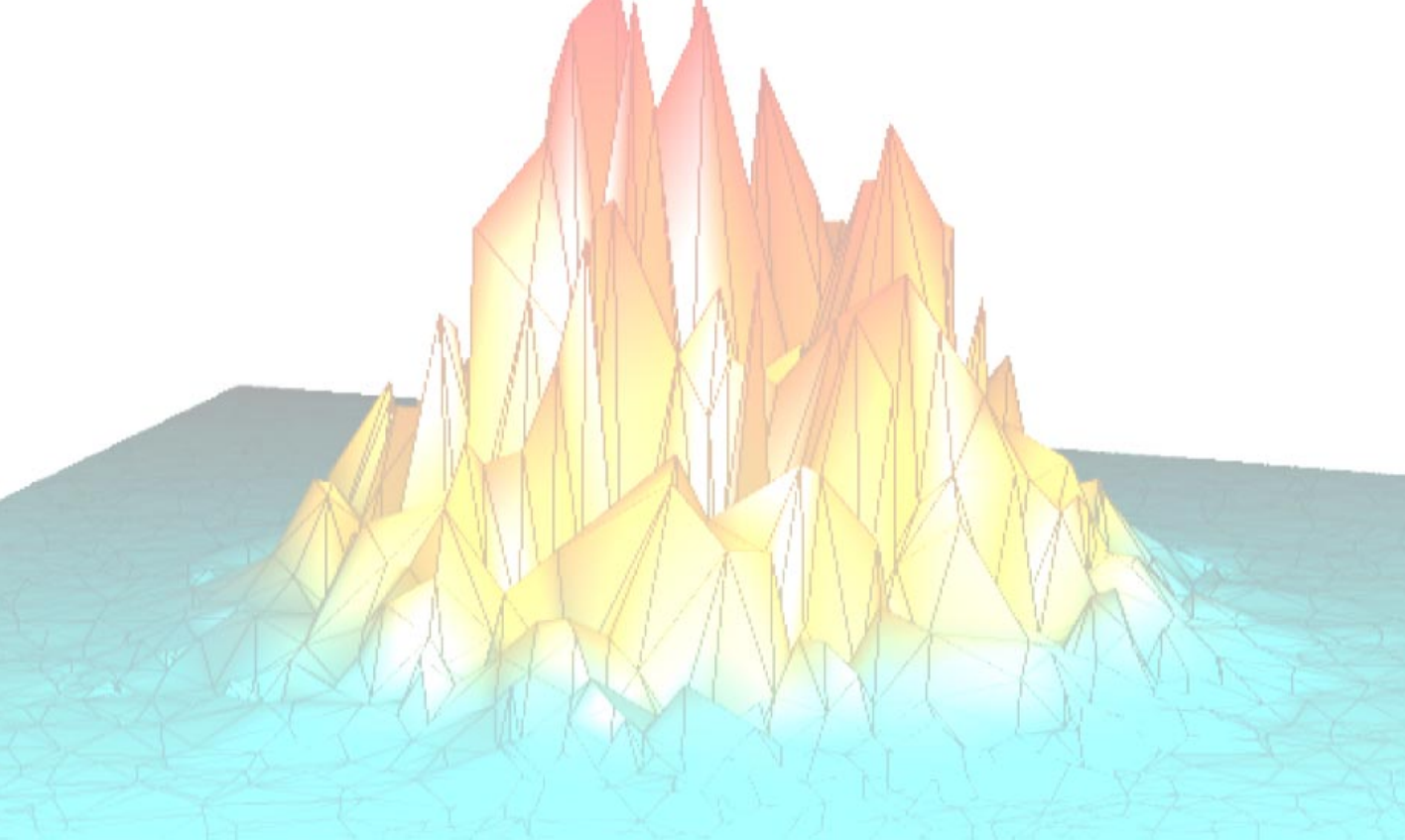
IDL is a complete computing environment for the interactive analysis and visualization of data. IDL integrates a powerful, array-oriented language with numerous mathematical analysis and graphical display techniques. Programming in IDL is a time-saving alternative to programming in FORTRAN or C—using IDL, tasks which require days or weeks of programming with traditional languages can be accomplished in hours. You can explore data interactively using IDL commands and then create complete applications by writing IDL programs.

Advantages of IDL include:

- IDL is a complete, structured language that can be used both interactively and to create sophisticated functions, procedures, and applications.
- Operators and functions work on entire arrays (without using loops), simplifying interactive analysis and reducing programming time.
- Immediate compilation and execution of IDL commands provides instant feedback and “hands-on” interaction.
- Rapid 2D plotting, multi-dimensional plotting, volume visualization, image display, and animation allow you to observe the results of your computations immediately.
- Many numerical and statistical analysis routines—including Numerical Recipes routines—are provided for analysis and simulation of data.
- IDL’s flexible input/output facilities allow you to read any type of custom data format. Support is also provided for common image standards (including BMP, GIF, JPEG, and XWD) and scientific data formats (CDF, HDF, and NetCDF).
- IDL widgets can be used to quickly create multi-platform graphical user interfaces to your IDL programs.
- IDL programs run the same across all supported platforms (Unix, VMS, Microsoft Windows, and Macintosh systems) with little or no modification. This application portability allows you to easily support a variety of computers.
- Existing FORTRAN and C routines can be dynamically-linked into IDL to add specialized functionality. Alternatively, C and FORTRAN programs can call IDL routines as a subroutine library or display “engine”.

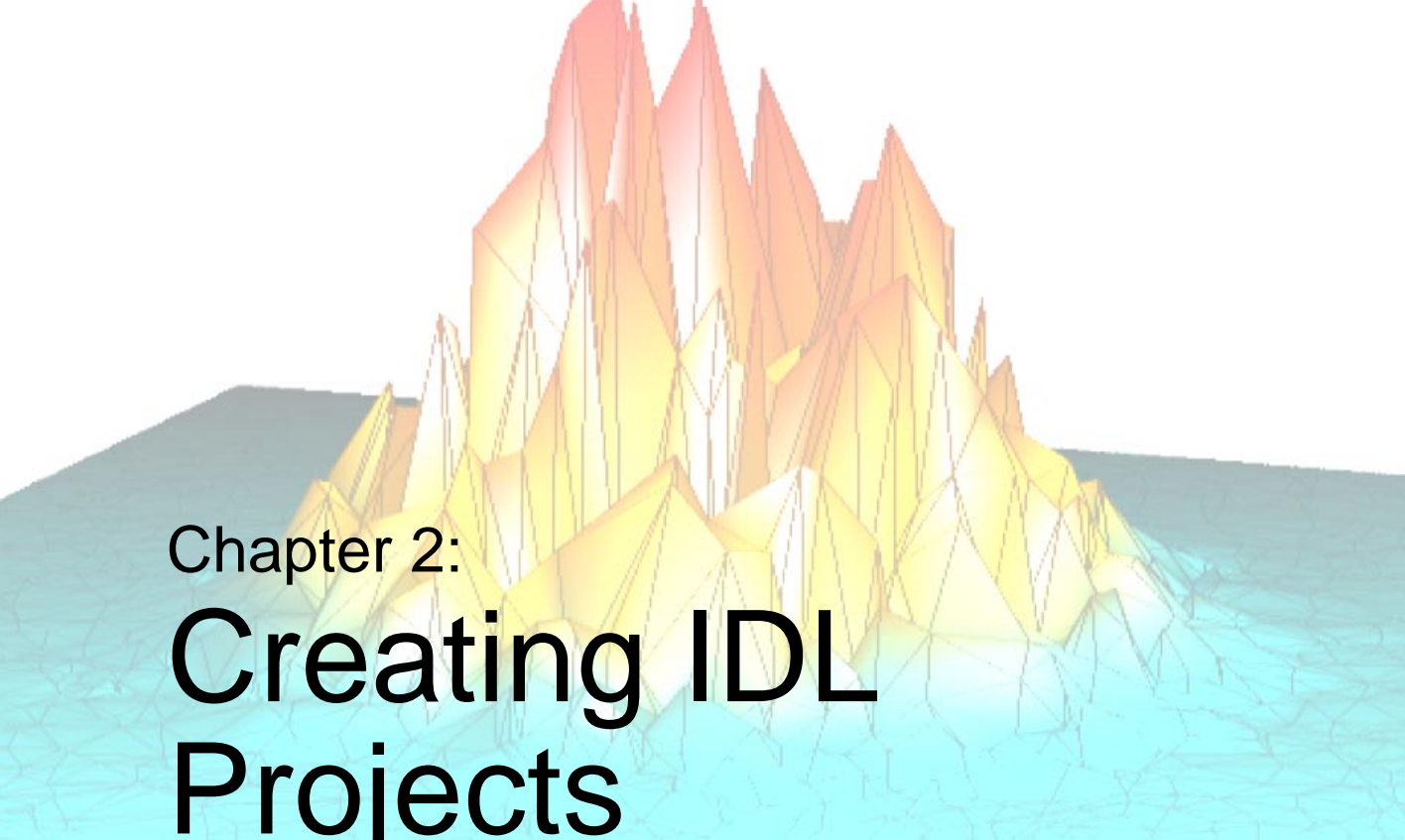






# ***Part I: How to Build Applications in IDL***





## Chapter 2:

# Creating IDL Projects

This chapter describes the following topics.

---

Overview .....	20	Selecting the Build Order .....	36
Where to Store the Files for a Project .....	22	Running an Application from a Project .....	41
Creating a Project .....	23	Compiling an Application from a Project ..	38
Opening, Closing, and Saving Projects .....	25	Building a Project .....	39
Adding, Moving, and Removing Files .....	26	Exporting a Project .....	42
Working with Files in a Project .....	29	About Developer's Kit Licenses .....	44
Setting the Options for a Project .....	33		

# Overview

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

## Access to all Files in Your Application

IDL Projects have an easy to use interface for grouping:

- IDL source code files (`.pro`)
- GUI files (`.prc`) created with IDL GUIBuilder
- Data files
- Image files
- Other files (help files, `.sav` files, etc.)

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

## Working with Files in Your Project

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

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

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

## Compiling and Running Your Application

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

code for IDL GUIBuilder (.prc) files, IDL Projects automatically prompt you to add them to your project.

## Exporting Your Applications

Once you have completed your application, you can quickly and easily create an IDL Runtime distribution or you can easily move your application to another platform or distribute your source code to colleagues by exporting your project. All your source code or compiled code (.sav files), IDL GUIBuilder files, data files, and image files are copied to a directory you specify.

## Example of a Project

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

# Where to Store the Files for a Project

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

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

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

where all of your source files (`.pro`) are in the `source` directory, IDL GUIBuilder files (`.prc`) are in the `gui` directory, data files are in the `data` directory, image files are in the `bitmaps` directory, and any other miscellaneous files are in the `other` directory. You can also create subdirectories under these directories.

---

**Note**

This examples names the directories the same as the folders that will hold your files in your project. You do not have to name your directories in this manner.

---

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

When a project's source files are exported, the files will be placed according to where they are in relation to the `.prj` file, keeping the directory structure intact whenever possible. All of the directories that are in the same directory as the `.prj` file will be recreated when an IDL Project is exported. If you have files that are stored outside of this hierarchy, they will be exported to the top-level directory. If, for example, one of your source files exists in `D:\otherproj`, when you export your project it will be placed in the top-level directory, in this case, `C:\myproject`.

For more information on exporting a project, see [“Exporting a Project”](#) on page 42.

# Creating a Project

To create a Project, complete the following steps:

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

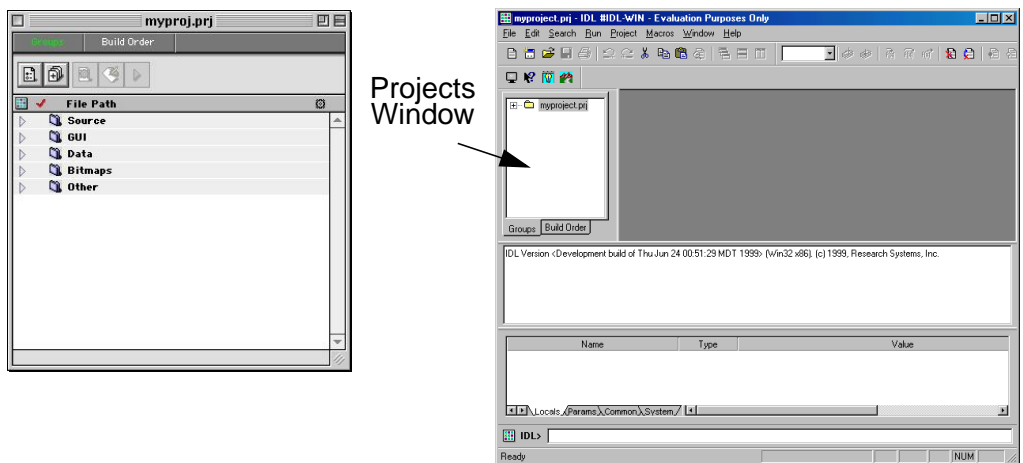


Figure 2-1: Projects Window for Macintosh (left) and Windows (right)

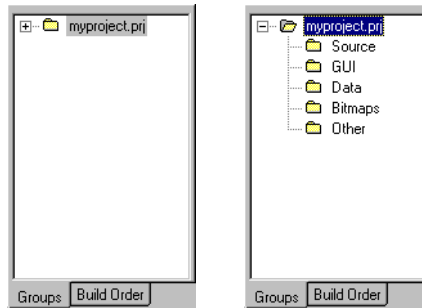
3. Save your new project. Select **File** → **Save Project**.

## Note

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

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

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



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

The following table describes the purpose for each group:

<b>Group</b>	<b>Description</b>
Source	Stores IDL source code files (.pro).
GUI	Stores GUI files (.prc) created using the IDL GUIBuilder.
Data	Stores any data files.
Bitmaps	Stores image files.
Other	Stores any other files that do not apply to the other groups.

*Table 2-1: Project Group Descriptions*



# Opening, Closing, and Saving Projects

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

## Opening Projects

To open a project, complete the following steps:

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

---

**Tip**

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

---

## Saving Projects

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

## Closing Projects

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

# Adding, Moving, and Removing Files

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

## Adding Files

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

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

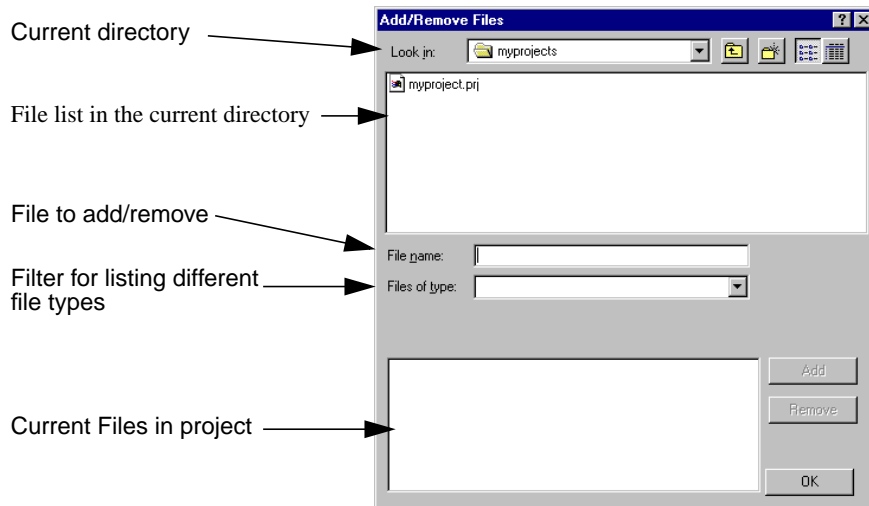


Figure 2-3: Add/Remove Dialog

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

### Tip

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

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

---

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

## Moving Files

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

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

### Note

---

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

---

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

### Note

---

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

---

## Removing Files

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

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

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

---

### Tip

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

---

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

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

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

---

### Tip

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

---

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

# Working with Files in a Project

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

## Editing a Source File

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

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

### Tip

---

You can also edit a .pro or .prc file by double-clicking on the filename. Also, on Windows you can drag the file from the Projects window to the IDL Editor window to open the file.

---

## Compiling a File

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

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

For more information on how to compile all the files in your project or just the files that have been recently modified, see [“Compiling an Application from a Project”](#) on page 38.

**Note**

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

**Testing a File**

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

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

For more information on running .prc files in test mode, see [“Running the Application in Test Mode”](#) on page 447.

**Setting the Properties of a File**

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

Property	Description
File name	The name of the file. (This field is read only.)
Path	The path of the file. (This field is read only.)
Group	The name of the group in which the file resides. (This field is read only.)
File Not Found	If the source file cannot be found, you can click this button to display a dialog for finding the path to the file.

*Table 2-2: File Properties*

Property	Description
Do not Compile	<p>Indicates whether or not to compile the file when running or building. For example, you may have included files for your main program that you do not want compiled. Selecting this check box indicates that you do not want this file compiled.</p> <p><b>Note</b> - You do not need to set this property for non-source files such as data files, image files, etc. These types of files will be automatically excluded from compilation.</p>
Export	<p>Indicates whether or not to export the file when exporting a project. Some files, such as data files that you need to use when creating your application, are files that you do not want to export. When checked, this file will be exported.</p>

*Table 2-2: File Properties*

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

1. Open your project. Select **File** → **Open Project** (on Windows and Motif) or **File** → **Open** (on Macintosh). Select the path and name of your project file.
2. Click on the plus sign (on Windows and Motif) or the expand arrow (on Macintosh) to expand the listing of the project files until you see the file you want to change.
3. Access the context menu by right-clicking (for Windows and UNIX) or Ctrl-clicking (for Macintosh) over the file for which you want to change the properties. Select **Properties** from the menu. The **File Properties** dialog is displayed.

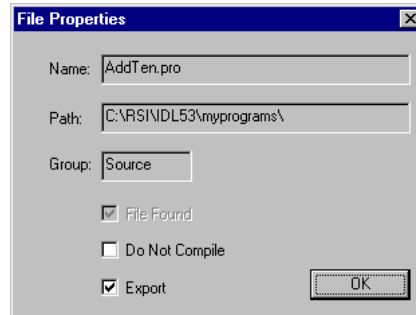


Figure 2-4: File Properties Dialog

4. Change any properties of the file.

---

**Note**

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

Also the Export option can also be selected/deselected by Ctrl-clicking over the file and selecting **Export** from the menu. If there is a check mark next to **Export**, the file will be exported.

---

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



## Setting the Options for a Project

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

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

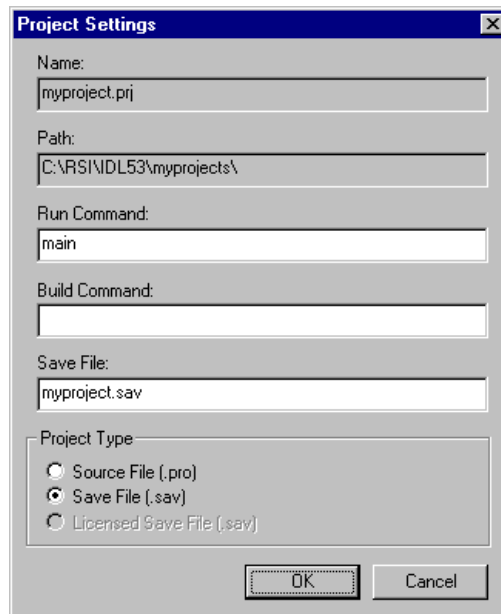


Figure 2-5: Project Settings Dialog

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

Option	Description
Name	Specifies the project name. <b>Note</b> - This field is read only.

Table 2-3: Project Options

Option	Description
Path	<p>Specifies the path of the project.</p> <p><b>Note</b> - This field is read only.</p>
Run Command	<p>Specifies the IDL command to run your application. The default is the name of the project. This can be any valid IDL command including <code>.sav</code> or <code>.pro</code> files (these can be files that are included or not included in your project.) Typically this is the main program in your application.</p> <p><b>Tip</b> - You can use the <code>%?</code> command stream substitution to call a dialog to enter a value or values to pass to the called program. For example, if you have a program named “main” and it requires the argument “x” to be passed to it, then you can enter the following for the Run Command:</p> <pre>main, %?(Enter the value for x, x)</pre> <p>For more information on how to run your application, see <a href="#">“Running an Application from a Project”</a> on page 41.</p>
Build Command	<p>Specifies the IDL command to build the application. The default is blank. If left blank, the files in the project are built according to the <a href="#">Execution File Format</a> specified and are compiled (if applicable) in the order specified under Build Order. For more information, see <a href="#">“Selecting the Build Order”</a> on page 36.</p> <p>You can enter any valid IDL command including <code>.sav</code> or <code>.pro</code> files. You can also enter a batch file using <code>@filename</code> in order to perform other operations (for example, running a Perl script on your source or data files before compiling. For more information on batch scripts, see the <i>Using IDL</i> manual.</p>
Save File	<p>Specifies the name of the save file to create when building your project. For more information on building a project, see <a href="#">“Building a Project”</a> on page 39.</p> <p><b>Note</b> - This field is grayed out if you have selected the <code>.pro</code> File Project Type.</p>

Table 2-3: Project Options

Option	Description
Execution File Format	<p>Specifies how the project will run or build. The available formats are:</p> <ul style="list-style-type: none"> <li>• Source File (.pro).</li> <li>• Save File (.sav).</li> <li>• Licensed Save File (.sav)</li> </ul> <p><b>Note</b> - The Licensed Save File option is grayed out if you do not have a Developer Kit license. For more information, see <a href="#">“About Developer’s Kit Licenses”</a> on page 44.</p> <p>For more information on building and running projects, see <a href="#">“Building a Project”</a> on page 39 or <a href="#">“Running an Application from a Project”</a> on page 41.</p>

*Table 2-3: Project Options*

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

---

**Note**

In addition to setting options for a project, you can also set the properties of a file. For more information, see [“Setting the Properties of a File”](#) on page 30.

---

## Selecting the Build Order

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

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

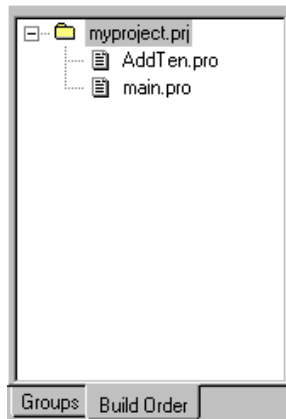
and file `AddTen.pro` contains:

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

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

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

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



*Figure 2-6: Build Order Window*

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

# Compiling an Application from a Project

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

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

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

---

## Note

If you have dependencies on the order in which your files are compiled, see [“Selecting the Build Order”](#) on page 36.

---

All the files in your project are now compiled. You can now run your application. For more information, see [“Running an Application from a Project”](#) on page 41.

## About IDL GUIBuilder Files

If you have included IDL GUIBuilder files in your IDL Project, the following

- On Windows, IDL GUIBuilder (.prc) files are not compiled automatically. You must select **File** → **Generate**. You are prompted whether or not to add the generated files (.pro file and event file) to your project.
- For UNIX and Macintosh, when you compile your IDL Project, IDL GUIBuilder (.prc) files are automatically compiled (the .pro and event files are generated) are automatically added to your project on Macintosh, but are not added to your project’s file listing for UNIX. On UNIX, you must add these files manually. They will be located in the same directory as the .prc file.

For more information on the IDL GUIBuilder, see [Chapter 17, “Using the IDL GUIBuilder”](#).

# Building a Project

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

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

For more information on resetting an IDL session, see [.FULL\\_RESET\\_SESSION](#) in the *IDL Reference Guide*. For more information on resolving undefined but referenced functions, see [RESOLVE\\_ALL](#) in the *IDL Reference Guide*.

- **Save File** — The IDL session is reset (all procedures, functions, main level variables, and common blocks are deleted from memory so that unwanted items are not included in your `.sav` file), all files in the project are compiled, all undefined but referenced functions and procedures are resolved, and all the functions and procedures are saved into the file you specified in the project's options.

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

- **Licensed Save File** — The IDL session is reset (all procedures, functions, main level variables, and common blocks are deleted from memory so that unwanted items are not included in your `.sav` file), all files in the project are compiled, all undefined but referenced functions and procedures are resolved, all the functions and procedures are saved into the file specified in the project's options, and embedded license information is added to the save file.

For more information on embedded license information, see [“About Developer's Kit Licenses”](#) on page 44.

---

## Note

For more information on project options, see [“Setting the Options for a Project”](#) on page 33.

---

To build your project, complete the following steps:

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

2. Select **Project** → **Build**. A dialog display confirming that you want to reset your session.

This will delete all procedures, functions, main level variables and common blocks from memory. If you have the save file option selected for your project, this will ensure that these items will not be included in your .sav file. If you have the source file option selected for your project, this will ensure that you have a clean environment in which to run and test your application.

3. Click **OK**.

Your project has been built.



# Running an Application from a Project

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

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

For more information on setting options for your project, see [“Setting the Options for a Project”](#) on page 33.

To run your application, complete the following steps:

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

## Exporting a Project

Once you have completed your application, you can quickly and easily create an IDL Runtime distribution or you can easily move your application to another platform or distribute your source code to colleagues by exporting your project. All your source code or compiled code (`.sav` files), IDL GUIBuilder files, data files, and image files are copied to a directory you specify.

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

- **Source File** — Your project’s source, IDL GuiBuilder, data, bitmaps, and any other files listed in your project will be exported along with your IDL Project file to a directory you specify so that you can move them to another platform. For information on how to set up a directory structure so that your IDL Project can find the source files after exporting, see [“Where to Store the Files for a Project”](#) on page 22.
- **Save File** — The `.sav` file for your project as well as data, bitmaps, and any other `.sav` files included in your project will be exported. You will also be given the option of exporting an IDL Runtime distribution for the platform to which you are exporting. For information on how to set up a directory structure so that all files will retain their relative paths after exporting, see [“Where to Store the Files for a Project”](#) on page 22.
- **Licensed Save File** — The `.sav` file (with an embedded license) for your project as well as data, bitmaps, and any other `.sav` files included in your project will be exported. You will also be given the option of exporting an IDL Runtime distribution for the platform you are exporting on. For information on how to set up a directory structure so that all files will retain their relative paths after exporting, see [“Where to Store the Files for a Project”](#) on page 22.

For more information on the options for a project, see [“Setting the Options for a Project”](#) on page 33. For more information on creating a `.sav` file with an embedded license, see [“About Developer’s Kit Licenses”](#) on page 44.

### Exporting Your Project’s Source Files

To export your project’s source files so that you can move them to another platform, complete the following steps:

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

2. Select **Project** → **Export**. The **Browse for Folder** dialog displays.
3. Select the folder to export the project and click **OK**.
4. A dialog is displayed asking if you want to export an IDL Runtime distribution with your `.sav` file. Select **No** to not include the distribution.

Your project has now been exported. When moving a project and its source files from one platform to another, there are a few items to be aware of:

- Project workspace information such as which files are open, etc. will not move from platform to platform.
- Problems with paths can occur if they are not relative paths. If you open a project and find that it cannot find the source file, you can fix this by changing the properties of the file. For more information, see [“Where to Store the Files for a Project”](#) on page 22 and [“Setting the Properties of a File”](#) on page 30.

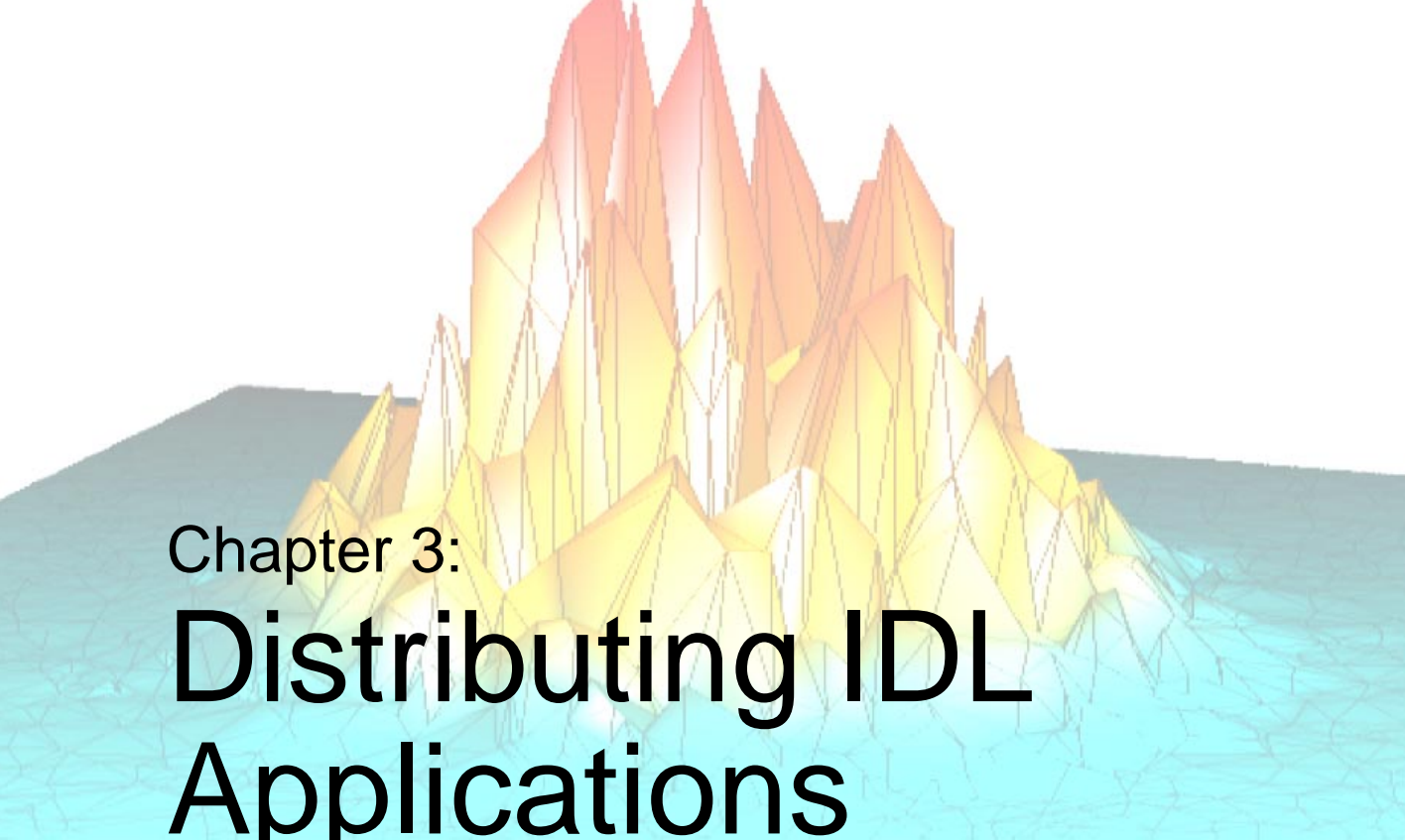
## Exporting `.sav` Files and an IDL Runtime Distribution

For more information on how to create a `.sav` file of your IDL application and an IDL Runtime distribution, see [“Creating Your Product Distribution Through Your IDL Project”](#) on page 51.

## About Developer's Kit Licenses

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

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



# Chapter 3: Distributing IDL Applications

This chapter describes the following topics.

---

Overview .....	46	For Applications That Use IDL DataMiner .	65
Creating Your Product Distribution Through Your IDL Project .....	51	For Applications That Use ActiveX .....	68
Customizing A Windows Distribution ....	55	Using the make_rt Script .....	69
Customizing a Macintosh Distribution ....	61	Adding IDL Files to the Distribution .....	73
Customizing A UNIX Distribution .....	64	Replacing the Licensing Dialog Image ....	76

# Overview

This chapter describes how to distribute your IDL application on Windows 95/98/NT, Macintosh, or UNIX platforms. Whether your IDL application uses `.sav` files, Callable IDL, or the IDL ActiveX control, you can easily distribute your IDL application using the procedures outlined in this section.

## IDL Applications

An IDL application is a `.sav` file or a series of `.sav` files as well as other data files, that you can distribute with an IDL Runtime distribution, a standard version of IDL, or by simply giving your user the IDL application to run on a version of IDL that they already have. When your user starts the program, IDL restores the `.sav` file you specify that contains the main program and executes the routines contained therein. If you are distributing a standard version of IDL or giving your application to a user who already has a standard version of IDL, you can also distribute `.pro` files since that user has the ability to compile programs.

## IDL Applications that Use Callable IDL

An IDL application that uses Callable IDL is an application written in another language (such as C, C++, Visual Basic, etc.) that calls IDL as a subroutine. You can distribute Callable IDL applications with an IDL Runtime distribution, a standard version of IDL, or by simply giving your user the IDL application to run on a version of IDL that they already have. For information on how to create an IDL Callable application, see the *External Development Guide*.

## IDL Applications that Use the IDL ActiveX Control

An IDL application that uses the IDL ActiveX control is an application written in another language (such as C, C++, Visual Basic, etc.) that uses ActiveX Controls to call IDL. The Microsoft Windows version of IDL includes an ActiveX Control that provides a powerful way to integrate all the data analysis and visualization features of IDL with other programming languages that support ActiveX control. You can distribute IDL ActiveX applications with an IDL Runtime distribution, a standard version of IDL, or by simply giving your user the IDL application to run on a version of IDL that they already have. For information on how to create an application that uses the IDL ActiveX control, see the *External Development Guide*.

## Using IDL DataMiner within Your Application

An IDL application that uses IDL DataMiner is an application (either using a `.sav` file, Callable IDL, or the IDL ActiveX control) that accesses data using the Open Database Connectivity (ODBC) interface. This allows IDL users to access and manipulate information from a variety of database management systems. Research Systems developed IDL DataMiner so that IDL users can have all the connectivity advantages of ODBC without having to understand the intricacies of ODBC or SQL (Structured Query Language). You can distribute applications that use IDL DataMiner with an IDL Runtime distribution, a standard version of IDL, or by simply giving your user the IDL application to run on a version of IDL that they already have. For information on how to create an application that uses IDL DataMiner, see the *IDL DataMiner Guide*.

### Your Application's Main `.sav` File

The main `.sav` file is the file that is restored and run when you start your IDL application. If you are creating your application using an IDL Project, all the information required to run your application will be automatically created when you export your project. Keep in mind that the `.sav` file that you want to automatically be restored and executed when IDL starts must have a procedure named “main” or a procedure with the same name as your `.sav` file without the extension.

### Secondary `.sav` Files and Other Data Files

If your IDL application requires that you provide data in IDL variables or if you want to distribute other procedures and functions that are not in your main `.sav` file, you will need to save the IDL variables, functions, and procedures in separate `.sav` files.

If you are creating your application using an IDL Project, you can include these `.sav` files in your project. You can also include other data files such as ASCII, binary, or image files that are required for your application in your project as well. When you export your project, these files will be included in the distribution you are creating. For more information on including other files in your IDL Project, see [“Where to Store the Files for a Project”](#) on page 22.

The `.sav` files included in your distribution can then be restored by a routine in your application. (Generally, the main procedure calls the RESTORE procedure to restore all secondary `.sav` files.)

---

**Note**

Before using the RESTORE procedure, make sure that you set !PATH system variable to a path that includes your secondary .sav files.

---

Keywords to the SAVE procedure allow you to save IDL variables, system variables, and common block definitions. For example, to save the definitions of all of the variables and common block definitions used in your compiled program in a file named `myvariables.sav`, use the command

```
IDL> SAVE, /VARIABLES, /COMM, FILENAME='myvariables.sav'
```

To restore the values of the variables saved in `myvariables.sav` from within your runtime application, include the line

```
RESTORE, 'myvariables.sav'
```

in your main procedure.

If you prefer not to include all of the IDL routines required by your application in your main .sav file, you can save them in one or more separate .sav files and use the RESTORE procedure in your main procedure to restore the routines:

```
RESTORE, 'extra_routines.sav'
```

---

**Note**

IDL Runtime does not compile .pro files. Because of this, you will need to create .sav files using the SAVE procedure in conjunction with the RESOLVE\_ALL procedure for any procedures or functions used by your application that are not included in the main .sav file.

---

## IDL Runtime vs. Standard IDL

You can distribute your IDL applications using an IDL Runtime distribution or using the standard IDL distribution.

### IDL Runtime

The Runtime version of IDL executes programs written using IDL, but does not provide access to the IDL command line or IDL Development Environment (IDLDE). It does not accept IDL commands interactively. You can create IDL programs and bundle them with IDL Runtime for distribution to users who do not have access to the full version of IDL.

IDL Runtime is appropriate for:



- Vertical-market packages developed in IDL but which appear to the user as stand-alone applications.
- Software designed for use by operators or technicians who do not need programmatic access to IDL's full range of analytical tools.
- Situations in which the developer does not want end-users to be able to modify functions written in the IDL language.
- Organizations with existing investments in IDL code, where some mixture of runtime and development IDL licenses may be cost effective.

## Standard IDL

The standard version of IDL is best suited to applications where the user needs access to the full scope of IDL's features. If your users need advanced analytical tools outside the scope of your application, you may choose to distribute your application using standard IDL. If you are distributing the standard version of IDL, contact your sales representative to purchase copies that you can distribute.

## How do I License My IDL Application?

IDL (both standard and IDL Runtime) is protected by one of the following means:

- **Software-Based Node-Locked License** — Use a software key to tie the license to the specific computer on which IDL will run. The software key is a license file generated by Research Systems or your local office or distributor based on a unique host ID from the machine to be licensed. This option is not available on the Macintosh platform.
- **Hardware-Based Node-Locked License** — Use a copy-protection device to tie the license to a specific computer (Windows and Macintosh platforms only). The device will not interfere with the normal functioning of your system or other software packages. The hardware key, or HASP, is a small plastic and metal device labeled HASP that plugs into your computer's parallel, ADB, or USB port.
- **Floating License** — Use a license file generated by Research Systems, your local office, or distributor based on a unique host ID from the machine or machines to be licensed. Floating licenses use a license server, which distributes available licenses to machines that request a license. This option is not available on the Macintosh platform.

**Note**

---

There are different hardware keys for the Runtime and standard versions of IDL. *On the Macintosh, both hardware keys cannot be installed on your machine at the same time.* This means that you or your users who have both products installed will need to have the correct hardware key installed while running the different versions of IDL. On Windows machines, you can install both hardware keys simultaneously.

---

You must decide which licensing method you will use to distribute your IDL applications. Research Systems strongly recommends that you distribute your IDL Runtime applications pre-licensed. This means that you will have to obtain the proper licensing information from Research Systems and individually license each copy of your application before delivering it to your end-user. This information can be included with your application. For more information, see the *Licensing Management Guide*.

If you would like more information on these licensing methods, please contact your Research Systems sales representative.

# Creating Your Product Distribution Through Your IDL Project

Once you have completed your application, you can quickly and easily create an IDL Runtime distribution that you can distribute to your users. All your compiled code (.sav files), IDL GUIBuilder files, data files, and image files are copied to a directory you specify.

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

- **Source File** — Your project's source, IDL GuiBuilder, data, bitmaps, and any other files listed in your project will be exported to a directory you specify so that you can move them to another platform. For more information on how to export source files, see [“Exporting Your Project's Source Files”](#) on page 42.
- **Save File** — The .sav file for your project will be exported. You will also be given the option of exporting an IDL Runtime distribution for the platform to which you are exporting.
- **Licensed Save File** — The .sav file (with an embedded license) for your project will be exported. You will also be given the option of exporting an IDL Runtime distribution for the platform you are exporting on.

For more information on the options for a project, see [“Setting the Options for a Project”](#) on page 33. For more information on creating a .sav file with an embedded license, see [“About Developer's Kit Licenses”](#) on page 44.

## Creating Your Application and IDL Runtime Distribution

To export your project's source files, complete the following steps:

---

### Note

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

---

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

**Note**

You must build the project before exporting. See “[Building a Project](#)” on page 39 for more information.

2. Select **Project** → **Export**. The **Browse for Folder** dialog is displayed.
3. Select the folder to which to export the project and click **OK**.
4. A dialog is displayed asking if you want to export an IDL Runtime distribution with your `.sav` file. Select **Yes** to include the distribution or **No** to not include the distribution. For information on how to add or remove files from the IDL distribution, see “[Adding IDL Files to the Distribution](#)” on page 73.
5. Specify where to copy the distribution files from.

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

For UNIX platforms, the distribution is taken directory from the installed version of IDL you are currently running.

For Macintosh platforms, a dialog displays that you can use to select the files you want to include in the distribution. If you want to create a standard distribution, select **Export**. For more information on customizing the distribution, see “[Adding IDL Files to the Distribution](#)” on page 73.

Your project has now been exported.

## The IDL Distribution for Windows

After you have exported your project and the IDL Runtime distribution, you will see the following in the directory you specified:

- Your application’s `.sav` file.
- Any other secondary `.sav` files or data files that you have included in your IDL Project.
- The `bin` and `resource` directories of the IDL Runtime distribution. The `bin` directory contains the subdirectory `bin.x86`, which contains all of the files necessary to run your application. The `resource` directory contains other files necessary to run IDL.

Within the `bin/bin.x86` directory are the executables for your application, which is named the same as your `.sav` file minus the extension (for example `your_sav_file.exe`), and the `idl.ini` file which describes information that IDL needs to start. Simply double-clicking the executable will start your application. All paths in the `idl.ini` file are relative so you can copy this directory tree to any other Windows platform and simply double-click the executable file to run your application.

For information on how to customize your IDL distribution, see [“Customizing A Windows Distribution”](#) on page 55.

## The IDL Distribution for Macintosh

After you have exported your project, you will see the following in the folder you specified:

- Your application’s `.sav` file.
- Any other secondary `.sav` files or data files that you have included in your IDL Project.
- The `bin` and `resource` folders.

The executable for your application is in the top-level folder that also contains your `main.sav` file and is named the same as your `main.sav` file. Simply double-click the executable to start your application. The executable uses information located in the resource for the file to start your application.

For information on how to customize your IDL distribution, see [“Customizing a Macintosh Distribution”](#) on page 61.

## The IDL Distribution for UNIX

After you have exported your project, you will see the following in the directory you specified:

- Your application’s `.sav` file.
- A script that executes your application that is named the same as your `.sav` file minus the extension.
- Any other secondary `.sav` files or data files that you have included in your IDL Project.
- The `bin` and `resource` directories.

You can simply execute the script that has been created to start your application.

For information on how to customize your IDL distribution, see [“Customizing A UNIX Distribution”](#) on page 64.

# Customizing A Windows Distribution

If you have exported your IDL Project with an IDL Runtime distribution or if you have used the `make_rt` script to create the distribution, the following sections describe ways in which you can customize your IDL application. You can perform these actions manually or through an installation script that you create.

## Note

---

There are a number of commercial applications available to help you build installers, for example InstallShield from InstallShield Software Inc. Research Systems has no connection with the companies that produce installation applications, and does not make any claims as to the applications' suitability.

---

## The IDL.INI File

IDL Runtime uses an initialization file—`idl.ini`—to supply software licensing information and program defaults. The `idl.ini` file is automatically created in the `bin\bin.x86` directory with settings that can be used to run your application when you export your IDL Project. If you have created an IDL distribution using the `make_rt` script, you will need to create the `idl.ini` file.

The `idl.ini` file is read when IDL starts up for the first time and its entries are converted into Windows Registry entries automatically. If registry entries already exist for an IDL installation in the same directory, a dialog will appear asking the user whether the existing entries should be overwritten. Even if you choose to have your customer license IDL the first time your application is run, you *must* supply an `idl.ini` file.

## Note

---

If your application's executable (created through your IDL Project), `idlde.exe`, or `idlrt.exe` is launched, then the `idl.ini` file will no longer be present. It will be renamed `idl.000` and a `reg.dat` file will be created. If this happens, you will need to rename the `idl.000` back to `idl.ini` and then distribute your package, without the `reg.dat` file.) In order to view it, since it is hidden, you may need to go to **View** → **Options** → **Show all files** in Windows Explorer and click **OK**.

---

The following is an example `idl.ini` file:

```
[IDL 5.3]
Cookie=25-XALP9CAF
```

```
HomeDir=..\..\
HelpPath=+..\..\help
InstallNum=314159
LicenseMethod=0
PortSetID=0
RSI Root=..\..\
RuntimeFile=..\..\myapp.sav
RuntimeIcon=myicon.ico
SearchPath=+..\..\
SiteNotice=My Notice
```

---

**Note**

If you have created an IDL Runtime distribution through IDL Projects, not all of the entries for the `idl.ini` file are included.

---

The fields have the following meanings:

**[IDL 5.3]**

The field heading specifies which version of IDL is in use. This heading should be the same as the heading used by the copy of IDL you use to create your IDL application.

**Cookie**

This field, along with `InstallNum` and `SiteNotice`, contain the hardware key license information for your IDL Runtime application. Note that these fields may contain different information for each installation.

If your application uses hardware licensing and you desire to “pre-license” your application, the fields `Cookie`, `InstallNum`, and `SiteNotice` should be set to the license information provided by Research Systems.

**HelpPath**

This field specifies which directories IDL Runtime will search for hypertext help files. The “+” symbol at the beginning of the string indicates that all subdirectories of the specified directory should be searched.

**HomeDir**

This field specifies the directory that contains the `bin` directory.



## InstallNum

This field, along with SiteNotice and Cookie, contain the hardware key license information for your IDL Runtime application. Note that these fields may contain different information for each installation.

If your application uses hardware licensing and you desire to “pre-license” your application, the fields Cookie, InstallNum, and SiteNotice should be set to the license information provided by Research Systems.

## LicenseMethod

The following line can be placed in the `idl.ini` file to cause IDL to use client/server licensing:

```
LicenseMethod=1
```

For this type of license, the `license.dat` file should be located in the `<idl_dir>\license` directory (where `<idl_dir>` is the directory that contains the IDL Runtime distribution tree). This type of license may require starting the license manager.

The default licensing method is Desktop (or hardware-based node-locked) licenses (HASP licenses). If desired, this could be forced by placing the following line in `idl.ini`:

```
LicenseMethod=0
```

## Note

---

When prompted with a dialog asking whether or not to import initialization preferences, you should answer **Yes**. Answering **No** may mean that you are not licensing IDL-Runtime.

---

## PortSetID

This field is used to describe the port number of the hardware key used for licensing. This field should always be set to 0.

## RSI Root

This field is used to specify the directory that contains the license directory for use with software-based node-locked licenses.

## RuntimeFile

This field should contain the name of the `.sav` file to be restored automatically when IDL Runtime starts. If this field is left blank and no `.sav` file is specified on the

command line, IDL Runtime will attempt to restore a file named `runtime.sav` in directory specified by the `HomeDir` field.

---

**Note**

If you specify a `.sav` file to be restored automatically, IDL Runtime will ignore any `.sav` file specified on the command line.

---

**RuntimeIcon**

This field should contain the path and name of the `.ico` file containing your application's custom icon.

**SearchPath**

This field specifies which directories IDL Runtime will search for `.sav` files. The “+” symbol at the beginning of the string indicates that all subdirectories of the specified directory should be searched.

**SiteNotice**

This field, along with `InstallNum` and `Cookie`, contain the hardware key license information for your IDL Runtime application. Note that these fields may contain different information for each installation.

If your application uses hardware licensing and you desire to “pre-license” your application, the fields `Cookie`, `InstallNum`, and `SiteNotice` should be set to the license information provided by Research Systems.

## Install the HASP Service

If you are using Hardware-Based Node-Locked License (or HASP licensing), you must install the HASP service.

Your IDL Runtime distribution tree contains an application (`hinstall.exe`) in the `bin\bin.x86` directory that will install or remove the HASP service. Use this application to install the service on your users' Windows systems. To do so, invoke `hinstall.exe` with the `/i` switch from the DOS prompt:

```
hinstall /i
```

as part of the installation process for your IDL application. Note that `hinstall.exe` will prompt the user to reboot the computer after installation of the HASP service.

To remove the HASP service, invoke `hinstall.exe` with the `/r` switch.

**Note**

Installing the HASP driver is not needed for floating licenses. For more information on installing floating licenses, see the *License Management Guide*.

## Creating Shortcuts/Start Menu Items

You can create shortcuts or Start menu items for your application. Use the following guidelines.

When you start your IDL Runtime application, it restores:

- a `.sav` file specified on the command line,
- a `.sav` file specified in the `idl.ini` file,
- or the file `runtime.sav` (if no other file is specified)

and calls the main procedure. This is:

- a procedure named `main` in the `.sav` file
- or a procedure with the same name as the `.sav` file

When the main procedure returns, IDL exits.

### Creating a Shortcut

To create a Shortcut that appears on your desktop, complete the following steps:

1. Right-click on your desktop and select **New** → **Shortcut**. The **Create Shortcut** dialog displays.
2. Enter the full path to your application's executable. For example:

```
c:\myapp\myapp.exe
```

This will call the `.sav` file specified in the `idl.ini` file. If you wanted to specify a `.sav` file in the command line, enter the following:

```
c:\myapp\myapp.exe myapp.sav
```

3. Click **Next**. The **Select a Title for the Program** dialog displays.
4. Enter the name you want to appear on the shortcut.
5. Click **Finish**.

## Creating a Start Menu Item

To create a Start Menu item, complete the following steps:

1. Select **Start** → **Settings** → **Taskbar and Start Menu...**. The **Taskbar Properties** dialog displays.
2. Select the **Start Menu Programs** tab.
3. Click **Add...**. The **Create Shortcut** dialog displays.
4. Enter the full path to your application's executable. For example:

```
c:\myapp\myapp.exe
```

This will call the `.sav` file specified in the `idl.ini` file. If you wanted to specify a `.sav` file in the command line, enter the following:

```
c:\myapp\myapp.exe myapp.sav
```

5. Click **Next**. The **Select a Title for the Program** dialog displays.
6. Enter the name you want to appear on the shortcut.
7. Click **Finish**.

# Customizing a Macintosh Distribution

If you have exported your IDL Project with an IDL Runtime distribution, the following sections describe ways in which you can customize your IDL application. You can perform these actions manually or through an installation script that you create.

---

## Note

There are a number of commercial applications available to help you build installers: these include Developer VISE from MindVision Software, Stuffit and Stuffit Installer Maker from Aladdin Systems, and DragInstall from Ray Sauers Associates, Inc. Research Systems has no connection with the companies that produce these applications, and does not make any claims as to the applications' suitability.

---

## Modifying The Resource for Your Application's Executable

IDL Runtime uses the resource of the executable to supply the name of the `.sav` file to restore on startup. When exporting a project and IDL Runtime distribution, the resource is automatically modified with the information necessary to run your application. This allows you to double-click on the IDL icon to run your application. You can modify the resource to change the default settings.

Resources are added using a resource file editor such as ResEdit or Resorcerer. The following instructions assume you are using ResEdit to add the necessary resource. If you are using a different editor, the exact actions may be slightly different.

## Prelicensing Your Application

---

## Note

Be sure to make your changes to a copy of IDL that is not licensed.

---

1. Double-click on the IDL icon to start IDL. Click **License**, and enter your runtime license key information into the License dialog.
2. Quit IDL.

## Changing the Main .sav File to Restore at Startup

1. Open the IDL application using ResEdit.

2. Open the STR# resources by double-clicking on the STR# icon.

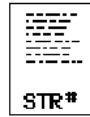


Figure 3-1: The STR# icon

3. Double-click on STR# ID 134. (The following figure shows the STR # ID in the title of the window.)

Enter the name of your application's main .sav file *without the .sav extension*. In the following figure, we have entered the name 'myapp' as an example.

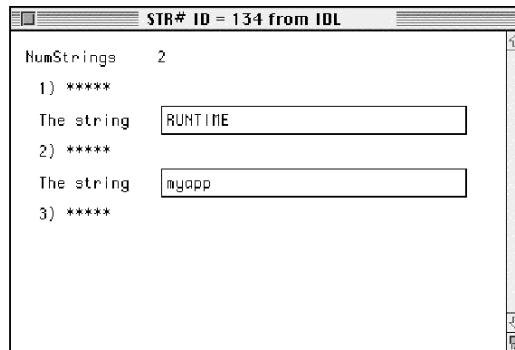


Figure 3-2: The ResEdit Resource String Dialog

### Note

The number of strings in the STR # 134 resource depends on the type of IDL license you are using. You may see more than two text fields—for example, if your license includes IDL DataMiner functionality, you will see a field that contains the string 'idl\_dm' in addition to the two shown above.

4. Choose **Save** from the File menu.
5. Quit ResEdit.

## Starting IDL Runtime Applications for Macintosh

There are three ways to start IDL Runtime for Macintosh:

- Double-click on the IDL Runtime icon. IDL Runtime will look for the file name specified in the STR # 134 resource (as described in “[Modifying The Resource for Your Application’s Executable](#)” on page 61) in the top level folder of the IDL runtime distribution, and attempt to restore the file. If it does not find the file, IDL will exit with an error message. This is the preferred method for launching an IDL runtime application.
- Drag the `.sav` file that contains the `MAIN` program onto the IDL Runtime icon. This method is also quite robust.
- Double-click on the `.sav` file that contains the `MAIN` program. This is the least-desirable method for starting a runtime application, because if more than one IDL executable (runtime or fully interactive) exists on a particular Macintosh, it is not certain that double-clicking on the `.sav` file will launch the correct executable. This may become a problem if:
  - You, as the IDL Runtime developer, have both runtime and non-IDL Runtime executables on your machine. In this situation, simply avoid double-clicking on the `.sav` file—drag it to the proper executable instead.
  - The IDL Runtime application end-user has other copies of IDL installed. Since it may not be possible to know in advance whether the end-user has other copies of IDL installed, it is up to you as a IDL Runtime application developer to inform your end-users that they should do one of the following:

Limit themselves to a single IDL executable. This may not be feasible if the end-user has an older version of IDL that cannot be upgraded to the same version as that used to produce the IDL Runtime application.

Explicitly drag the runtime application’s `.sav` file to the proper IDL Runtime executable. This will work in all cases.

Double-click on the IDL Runtime executable icon to start the application. Note that this method requires that the name of the `.sav` file be specified in the STR # 134 resource of the IDL executable, or (if no name is specified in the string resource) that the `.sav` file be named `runtime.sav`. In either case, the `.sav` file must be located in the `lib` folder in the IDL Runtime distribution.

# Customizing A UNIX Distribution

If you have Exported your IDL Project with an IDL Runtime distribution or if you have used the `make_rt` script to create the distribution, the following section describe ways in which you can customize your IDL application. You can perform these actions manually or through an installation script that you create.

## IDL Runtime Command Line Options for UNIX

IDL Runtime for UNIX is started using the `idl` command. You can invoke this command at the command line or create a script containing the `idl` command. If you have Exported your IDL Project, a script has been automatically created for you.

When you start your IDL Runtime application, it restores:

- a `.sav` file specified on the command line,
- or the file `runtime.sav` (if no other file is specified)

and calls the main procedure. This is:

- a procedure named `MAIN` in the `.sav` file
- or a procedure with the same name as the `.sav` file

When the main procedure returns, IDL exits.

The following command will start IDL Runtime and attempt to restore a file named `runtime.sav`, located in the current directory:

```
% idl -rt
```

The following command will start IDL Runtime and attempt to restore a file named `myapp.sav`, located in the directory `/usr/local/rsi/idl`:

```
% idl -rt=/usr/local/rsi/idl/myapp.sav
```

If the `.sav` file is located in the current directory, the full path name need not be supplied.



# For Applications That Use IDL DataMiner

If your application uses IDL DataMiner, you will need complete the following tasks:

## If Your Application Uses IDL DataMiner for Windows

1. Before creating your IDL Runtime distribution, add the files listed in the `manifest_aux.txt` file to the manifest file. For more information on how to modify the manifest file, see [“Adding IDL Files to the Distribution”](#) on page 73.
2. After creating your IDL Runtime distribution, you will need to move files from the `bin\bin.x86` directory. Move the following .DLLs to the `WINDOWS/SYSTEM` directory for Windows 95/98 or to the `WINDOWS/SYSTEM32` directory for Windows NT.

<code>dmbas13.dll</code>	<code>dmut13.dll</code>	<code>ivdm.lic</code>
<code>odbc32.dll</code>	<code>odbccp32.dll</code>	<code>odbcint.dll</code>
<code>odbcad32.exe</code>	<code>odbccr32.dll</code>	<code>odbcinst.cnt</code>
<code>odbcinst.hlp</code>	<code>odbctrac.dll</code>	<code>mtxdm.dll</code>
<code>dmdrv13.hlp</code>	<code>dmdrv13.cnt</code>	<code>dmflt13.dll</code>

3. You will also need to move .DLLs for any the drivers you require to the `WINDOWS/SYSTEM` directory for Windows 95/98 or to the `WINDOWS/SYSTEM32` directory for Windows NT. The following are the drivers for Windows 95/98/NT:

### Files for MS SQL Server

- `dbnmpntw.dll`
- `drvssrvr.hlp`
- `sqlsrv32.dll`

### Files for Informix

- `dminf13.dll`
- `dminf13.hlp`

**Files for Oracle 7**

- dmor713.dll
- dmor713.hlp

**Files for Oracle 8**

- dmor813.dll
- dmor813.hlp

**Files for Sybase**

- dmsyb13.dll
- dmsyb13.hlp

**Files for Text Files**

- dmtxt13.dll
- dmtxt13.hlp

4. You will also need to create registry entries for these files. The `registry.txt` file located in the `bin/make_rt` directory contains the necessary information to create these entries.

## For Macintosh

1. Before creating your IDL Runtime distribution, copy the ODBC folder located in the *RSI-directory* to the IDL53 directory where *RSI-directory* is the directory in which you have installed IDL.
2. Choose **Project** → **Export**. The export dialog is displayed.
3. Add all the files the ODBC directory that you have just copied by clicking the ODBC directory. A check mark is displayed next to the folder indicating that it will be exported.
4. When you are installing your application, you will need to move files from the ODBC directory of the IDL Runtime distribution.

Move the following files to the System:Extensions folder:

- ODBC Configuration Manager PPC
- ODBC Cursor Library PPC
- ODBC Driver Manager PPC

- ODBC 2.x Bridge PPC
- ODBC Trace Library PPC

Move the following file to the System:Control Panels folder:

- ODBC Setup PPC

Move the following file to the System:Preferences folder:

- dmodbc.lic

Move any of the following files to the Systems:Extensions:ODBC folder that your application requires:

- Research Systems 3.02 Flat Lib
- Research Systems 3.02 Base Lib
- Research Syste 3.02 Text Driver
- Research Syst 3.02 dBase Driver
- Research Sys 3.02 Utilities Lib
- Research Sys 3.02 Sybase Driver
- Research Sys 3.02 Oracle Driver
- Research S 3.02 FoxPro DB Driver

## For UNIX

1. Before creating your IDL Runtime distribution, add the files listed in the `manifest_aux.txt` file to the manifest file. For more information on how to modify the manifest file, see [“Adding IDL Files to the Distribution”](#) on page 73.
2. You must modify the `odbc.ini` file to include information about the drivers you are using. This file is located in the `resource/dm/<OS_NAME>` directory of the distribution tree you have just created. After modifying this file, it must be placed in each user’s home directory. For details on the modifications you must make to the `odbc.ini` file, see the *IDL DataMiner* manual.

# For Applications That Use ActiveX

If you're creating a distribution for an application that uses the IDL ActiveX control on Windows platforms, you must complete the following steps:

1. Before creating your IDL Runtime distribution, add the files listed in the `manifest_aux.txt` file to the manifest file. For more information on how to modify the manifest file, see [“Adding IDL Files to the Distribution”](#) on page 73.
2. Install the `idldrawx2.ocx` file which is now in the `bin.x86` directory of your distribution tree in the `SYSTEM` subdirectory of the `WINDOWS` directory (on Windows NT systems, the files should be installed in the `SYSTEM32` directory).
3. Register the `idldrawx2.ocx` file during your installation process. For example, you can do this using the `regsvr32.exe` executable (for more information, refer to your Microsoft Windows documentation) or you can perform this task using calls from the product you are using to create your installation script.

## Using the `make_rt` Script

The `make_rt` script is used to create an IDL Runtime distribution. On Macintosh, this process is done through a dialog. This process can be used to create IDL Runtime distributions for use with IDL Callable or IDL ActiveX applications. This script is also called during the Exporting of an IDL project. If you choose, you can run this script outside of projects to create your IDL Runtime distribution.

---

### Tip

It is recommended that you use the Exporting of an IDL Project to create your IDL Runtime distribution instead of using the `make_rt` script. For more information on how to create an IDL Runtime distribution through IDL Projects, see [Chapter 2, “Creating IDL Projects”](#).

---

If you are using the `make_rt` script to create your IDL Runtime distribution, you must complete the following tasks that Exporting an IDL Project does for you automatically:

- Create the IDL Runtime distribution using the `make_rt` script. If you are creating a distribution for an application that uses the IDL ActiveX control, or an application that uses IDL DataMiner, see [“Adding IDL Files to the Distribution”](#) on page 73 for information on additional files you’ll need to include. You can also customize the distribution to include other files in the standard IDL distribution, for more information see [“Adding IDL Files to the Distribution”](#) on page 73.
- Place your application’s main `.sav` file, along with any `.sav` files containing variable data or additional routines, in the distribution tree you have created.
- Create `.sav` files for any `.pro` files from the IDL distribution that are used by your application and place them in the distribution tree you have just created. Alternatively, these routines can be compiled and saved in your application’s main `.sav` file.

---

### Note

Before using the RESTORE procedure to restore any secondary `.sav` files, make sure that you set `!PATH` system variable to a path that included your secondary `.sav` files.

---

## Using the make\_rt Script for Windows

To create a distribution on Windows using the `make_rt` script, complete the following steps:

### Note

The `make_rt.exe` is not supported on Windows NT for Alpha platforms.

1. Select **Start** → **Run**. The Run dialog is displayed.
2. Enter the command syntax based on the following:

```
rsi-directory\bin\make_rt\make_rt.exe [source] dest manifest savefile mode
```

where *rsi-directory* is the directory in which you installed IDL. The following table describes each parameter:

Parameter	Description
<i>source</i>	The CD-ROM drive containing your IDL product CD-ROM to copy the distribution source files. For example, if your CD-ROM drive is E:, then the path would be "E:". If not specified, you will be prompted.
<i>dest</i>	The full path to the destination directory to contain the copied distribution.
<i>manifest</i>	The full path and filename containing the list of files to copy. This is the <code>manifest_rt.txt</code> file in the <i>rsi-directory</i> \bin\make_rt directory. For more information, see <a href="#">“Adding IDL Files to the Distribution”</a> on page 73.
<i>savefile</i>	The name of the resulting IDL Runtime executable. The IDL Runtime executable is typically named “ <code>idlrt.exe</code> ” but you can specify the name of your product or the same name as your <code>.sav</code> file. For IDL Callable or IDL ActiveX control applications, we suggest you use “ <code>idlrt</code> ”. Enter the name without any extension.

Parameter	Description
<i>mode</i>	The type of distribution you are creating. Valid values are: rt — Creates an IDL Runtime distribution for use with all IDL applications, including IDL Callable applications and ActiveX applications. em — Creates an IDL Runtime distribution for use with an embedded <code>.sav</code> file. For more information, see <a href="#">“About Developer’s Kit Licenses”</a> on page 44.

Table 3-1: *make\_rt.exe* Parameters

3. Click **OK**.

## For Macintosh

On Macintosh, the distribution is created through a dialog where you can choose the files to export. To start the dialog, choose **Project** → **Export**. For more information on creating a distribution, see [“Adding Files on Macintosh Platforms”](#) on page 74.

## For UNIX

To create a distribution on Windows, run the `make_rt` script located in `rsi-directory/bin/make_rt` script where *rsi-directory* is the IDL installation directory. The following is the syntax for the `make_rt` script:

```
make_rt [source] dest manifest savefile mode
```

The following table describes each parameter of the syntax:

Parameter	Description
<i>source</i>	The full path to the source directory containing the IDL distribution. This is the path to the IDL <code>bin</code> directory. For example, if you’ve installed in the default location, this would be <code>/usr/local/rsi/idl_5.3</code> . If not specified, you will be prompted.

Table 3-2: *make\_rt* Parameters

Parameter	Description
<i>dest</i>	The full path to the destination directory to contain the copied distribution.
<i>manifest</i>	The full path and filename containing the list of files to copy. This is the <code>manifest_rt.txt</code> file in the <code>rsi-directory/bin</code> directory. For more information, see <a href="#">“Adding IDL Files to the Distribution”</a> on page 73.
<i>savefile</i>	The name of the script to be created in the <i>dest</i> directory. This script will start IDL Runtime and attempt to restore a <code>.sav</code> file named <i>savefile.sav</i> in your <i>dest</i> directory. You may want to specify the name of your product or the same name as your <code>.sav</code> file. Enter the name without any extension.
<i>mode</i>	The type of distribution you are creating. Valid values are:  rt — Creates an IDL Runtime distribution for use with all IDL applications including IDL Callable applications and ActiveX applications.  em — Creates an IDL Runtime for use with an embedded <code>.sav</code> file. For more information, see <a href="#">“About Developer’s Kit Licenses”</a> on page 44.

Table 3-2: *make\_rt* Parameters



## Adding IDL Files to the Distribution

The files that are exported for an IDL Runtime distribution are defined in a manifest file. This file is used when you export a Project and include the IDL Runtime distribution. The files that are included in the manifest file are the minimum IDL files you need to create an IDL Runtime distribution. You can modify this file to:

- Add the resource files such as high-resolution maps that are not currently included in the manifest.
- Add files required for creating an IDL ActiveX distribution.
- Add files required for an IDL application that uses IDL DataMiner.
- Add any other files included in the IDL distribution that are not included in the manifest file.
- Remove any files that are included in the manifest but are not required for your application.

---

### Note

Only files in the IDL distribution can be added to the manifest. If you have other files you want to be included in your application but are not in the IDL distribution, you can add them to your IDL Project or manually copy them into the distribution after it has been completed.

---

## Adding Files on Windows and UNIX Platforms

The manifest is located in *rsi-directory/bin/make\_rt/manifest\_rt.txt* file for Windows platforms and *rsi-directory/bin/manifest\_rt.txt* for Motif platforms where *rsi-directory* is the installation directory for IDL.

To modify the manifest file to include other files, complete the following steps:

1. Open the *manifest\_rt.txt* in any text editor.
2. If you're creating a distribution for use with an IDL ActiveX application or an IDL application that uses IDL DataMiner, you need to add the files listed in the *manifest\_aux.txt* file located in the same directory as the *manifest\_rt.txt* file.
3. Add the path and filename of any other files you want to include to the list of files to export. Make sure that the path is relative to the *rsi-directory*.

**Note**

On Windows, files are copied from your IDL product CD-ROM. In this case, *rsi-directory* is the `idl53` directory on your IDL product CD-ROM. Note also that on the CD-ROM, there are multiple subdirectories under the `bin` directory for the different Windows platforms.

4. Make sure that you have not included any blank lines in the file.
5. Save the file.

## Adding Files on Macintosh Platforms

On the Macintosh platform, when you choose to export an IDL Runtime distribution by selecting **Project** → **Export**, a dialog is displayed which you can use to choose the files you want to include in your IDL Runtime distribution. Complete the following steps to add files to your distribution:

1. Select the files you want to include in the manifest by clicking on them. You can only add files that are in the IDL 5.3 folder. A check mark appears next to a file that is to be included. The files that are already checked are the minimum IDL files you need to create an IDL Runtime distribution. A check next to a folder indicates the entire contents of the folder will be included in the distribution. A dash next to a folder means that only some of the files (files with check marks next to them) will be included in the distribution.
2. Select the graphic to use as the application's icon. You can copy and graphic you want to use and then select the icon. Choose **Edit** → **Paste** to insert the graphic.
3. Select the Preferred and Minimum Size for memory to be used by your application. This is listed in Kilobytes. The default values listed are the values that are currently set for IDL.
4. Click **Export**.

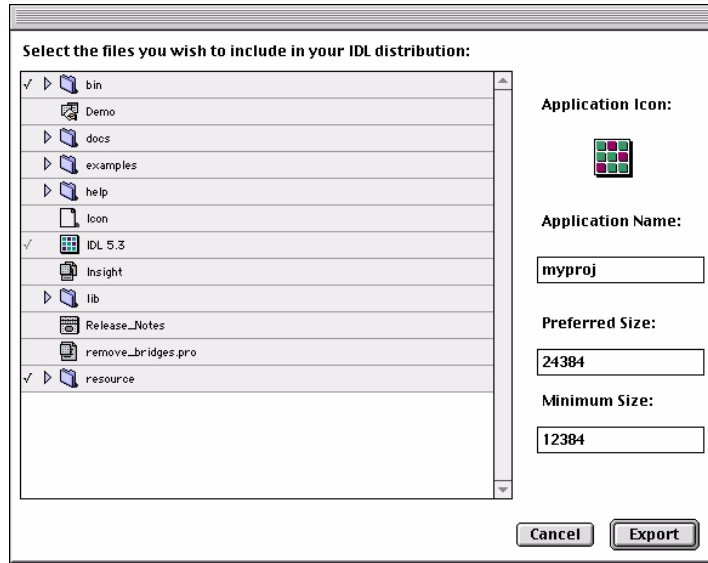


Figure 3-3: Macintosh Distribution File List

## Replacing the Licensing Dialog Image

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

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

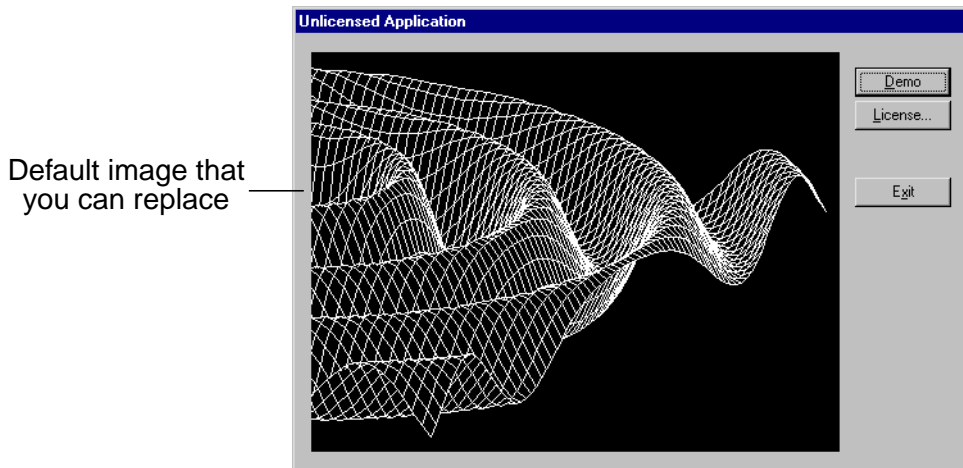


Figure 3-4: Unlicensed Application Dialog for Windows

### Replacing the Image for Windows Callable Applications

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

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

You must call the `IDL_SetValue` routine prior to the `IDL_Win32Init()` call which initializes IDL. `pvValue` may be either a string containing the path of a `.bmp` file or a bitmap resource defined in your IDL Callable application. The `id` attribute is always `IDL_VAL_DEMODLG_BITMAP` as defined in `export.h`.

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

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

```
IDL_SetValue(IDL_VAL_DEMODLG_BITMAP, (void*) bitmapFile);
```

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

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

where `IDB_BITMAP1` is a constant in your application.

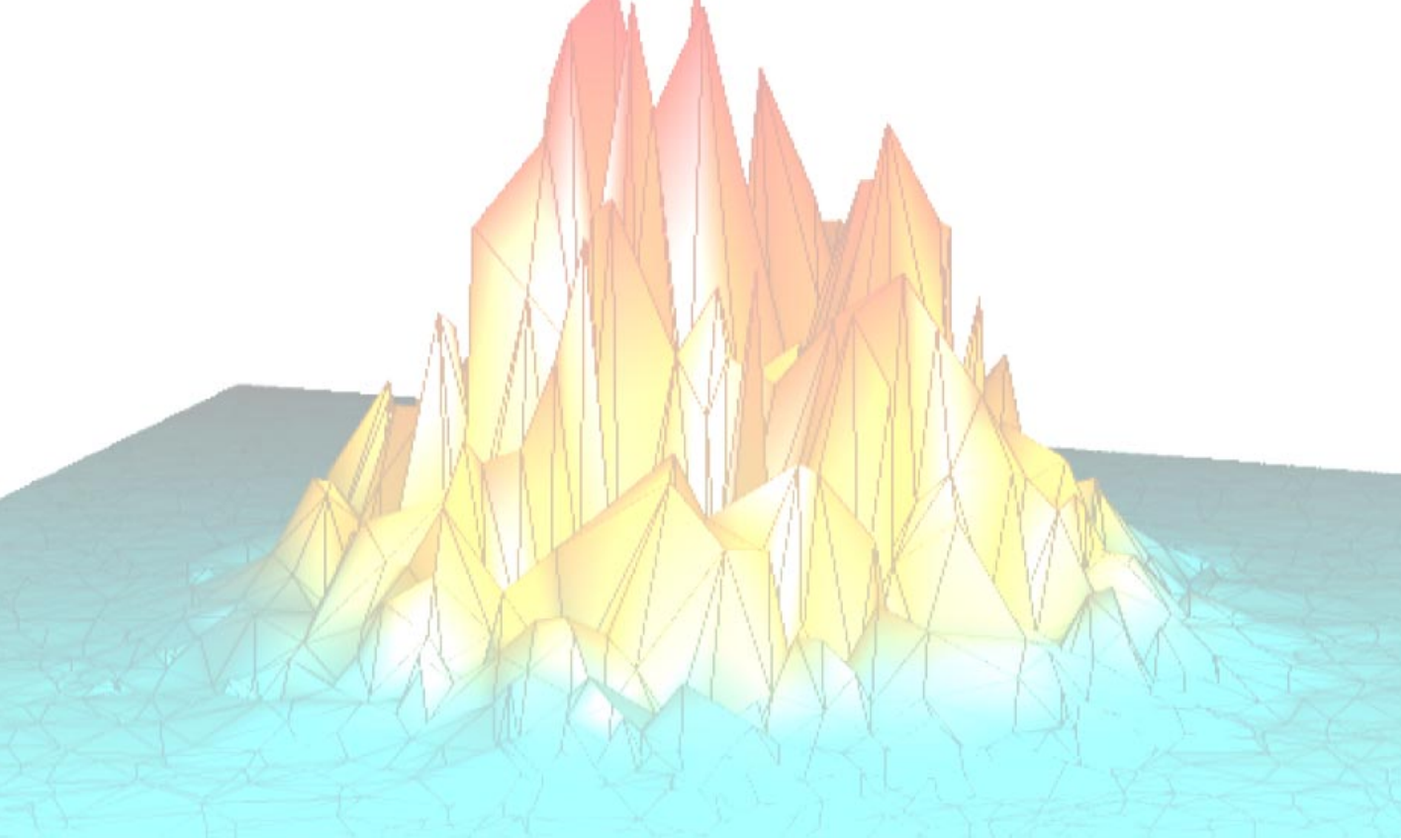
## Replacing the Image for Macintosh Callable Applications

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

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

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





# ***Part II: Components of IDL***







## Chapter 4:

# The Structure of the IDL Language

This chapter provides a brief overview of IDL's data types and language constructs. The following topics are covered in this chapter:

---

Data Types .....	82	Arrays .....	90
Numeric Constants .....	84	Structures .....	93
String Constants .....	86	Variables .....	96
Type Conversion Functions .....	87	System Variables .....	99

# Data Types

The IDL language is *dynamically typed*. This means that an operation on a variable can change that variable's type. In general, when variables of different types are combined in an expression, the result has the data type that yields the highest precision.

For example, if an integer variable is added to a floating-point variable, the result will be a floating-point variable.

## Basic Data Types

In IDL there are twelve atomic data types, each with its own form of constant. The basic data types and their syntax are defined in the following table:

Data Types	Syntax	Definition
Byte	<i>nB</i>	8-bit unsigned integer ranging in value from 0 to 255
Integer	<i>n</i> or <i>nS</i>	16-bit signed integer ranging from -32,768 to +32,767
Unsigned Integer	<i>nU</i> or <i>nUS</i>	16-bit unsigned integer ranging from 0 to 65535.
Longword	<i>nL</i>	32-bit signed integer in the range of $\pm$ two billion, $2 \times 10^9$
Unsigned Longword	<i>nUL</i>	32-bit unsigned integer in the range 0 to four billion.
64-bit Integer	<i>nLL</i>	64-bit integer ranging from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
Unsigned 64-bit Integer	<i>nULL</i>	Unsigned 64-bit integer ranging from 0 to 18,446,744,073,709,551,615.
Floating Point	<i>n.n</i>	32-bit, single-precision, floating-point number in the range of $\pm 10^{38}$ (IEEE)

Table 4-1: Basic Data Types

Data Types	Syntax	Definition
Double-Precision	$n.nD$	64-bit, double-precision, floating-point number in the range of $\pm 10^{308}$ (IEEE)
Complex	$\text{COMPLEX}(n.n, n.n)$	real-imaginary pair of single-precision, floating-point numbers
Double-Precision Complex	$\text{DCOMPLEX}(n.n, n.n)$	real-imaginary pair of double-precision, floating-point numbers
String	' <i>ssss</i> '	sequence of characters, from 0 to 32,767 characters in length

Table 4-1: Basic Data Types

In addition to the twelve basic data types, IDL also supports the following:

- Structures: Aggregations of data of various types. Structures are discussed in [Chapter 7, “Structures”](#).
- Pointers: A reference to a dynamically-allocated *heap variable*. Pointers are discussed in [Chapter 11, “Pointers”](#).
- Object References: A reference to a special heap variable that contains an IDL object structure. Object references are discussed in [Chapter 12, “Object Basics”](#).

# Numeric Constants

This section briefly discusses the features of IDL's numeric constants. The syntax of numeric constants is described further in [Chapter 5, "Constants"](#).

## Integer Constants

Numeric constants of different types can be represented by a variety of forms. Integer constants can be decimal, hexadecimal or octal. Each of these radices can either be of byte, integer, long, or 64-bit long type. The absolute values of integer constants are given in the following table:

Type	Absolute Value Range
Byte	0 – 255
Integer	0 – 32767
Unsigned Integer	0 – 65535
Long	0 – (2 <sup>31</sup> - 1)
Unsigned Long	0 – (2 <sup>32</sup> - 1)
64-bit Long	0 – (2 <sup>63</sup> - 1)
Unsigned 64-bit Long	0 – (2 <sup>64</sup> - 1)

*Table 4-2: Absolute Value Range Of Integer Constants*

Integers specified without one of the B, S, L, or LL codes are automatically promoted to an integer type capable of holding them. For example, 4000 is promoted to longword because it is too large to fit into an integer. Any numeric constant can be preceded by a plus (+) or minus (-) sign.

## Floating-Point and Double-Precision Constants

Floating-point and double-precision constants can be expressed in either conventional or scientific notation. Any numeric constant that includes a decimal point is a floating-point or double-precision constant.

## Complex Constants

Complex constants contain a real and an imaginary part, both of which are single- or double-precision, floating-point numbers. The imaginary part can be omitted, in which case it is assumed to be zero. The form of a complex constant is as follows:

```
COMPLEX(REAL_PART, IMAGINARY_PART)
```

*or*

```
COMPLEX(REAL_PART)
```

For example, `COMPLEX(1,2)` is a complex constant with a real part of one, and an imaginary part of two. `COMPLEX(1)` is a complex constant with a real part of one and a zero imaginary component. To extract the real part of a complex expression, use the `FLOAT` function. The `ABS` function returns the magnitude of a complex expression, and the `IMAGINARY` function returns the imaginary part.

# String Constants

An IDL string is a sequence of characters from 0 to 65,535 characters in length, enclosed by apostrophes (') or quotes ("). The value of the constant is simply the characters appearing between the leading delimiter (' or ") and the next occurrence of the *same* delimiter. A double apostrophe (')') or quote (" ") is considered to be the null string; a string containing no characters. An apostrophe or quote can be represented within a string by two apostrophes or quotes; e.g., 'Don''t' returns Don't. This syntax often can be avoided by using a different delimiter; e.g., "Don't" instead of 'Don''t'.

Strings have dynamic length (they grow or shrink to fit), and there is no need to declare the maximum length of a string prior to using it. As with any data type, string arrays can be created to hold more than a single string. In this case, the length of each individual string in the array depends only on its own length and is not affected by the lengths of the other string elements.

Features of IDL string constants are described further in [Chapter 9, “Strings”](#).

# Type Conversion Functions

The conversion functions are as follows:

Function	Description
STRING	Convert to string
BYTE	Convert to byte
FIX	Convert to 16-bit integer, or optionally other type
UINT	Convert to 16-bit unsigned integer
LONG	Convert to 32-bit longword
ULONG	Convert to 32-bit unsigned longword
LONG64	Convert to 64-bit integer
ULONG64	Convert to 64-bit unsigned integer
FLOAT	Convert to floating-point
DOUBLE	Convert to double-precision floating-point
COMPLEX	Convert to complex value
DCOMPLEX	Convert to double-precision complex value

*Table 4-3: Type Conversion Functions*

These functions are useful in many instances, such as forcing the evaluation of an expression to a certain type, outputting data in a mode compatible with other programs, etc.

The following table illustrates several uses of type conversions:

Operation	Results
FLOAT(1)	1.0
FIX(1.3 + 1.7)	3
FIX(1.3) + FIX(1.7)	2
FIX(1.3, TYPE=5)	1.3000000
BYTE(1.2)	1
BYTE(-1)	255 (Bytes are modulo 256)
BYTE('01ABC')	[48, 49, 65, 66, 67]
STRING([65B, 66B, 67B])	'ABC'
FLOAT(COMPLEX(1, 2))	1.0
COMPLEX([1, 2], [4, 5])	[COMPLEX(1,4),COMPLEX(2,5)]

*Table 4-4: Uses of Type Conversion Functions*

Type conversion between strings and bytes is a special case. The result of the BYTE function applied to a string or a string array is a byte array containing the ASCII codes of the characters of the string. Converting a byte array with the STRING function yields a string array or scalar with one less dimension than the argument.

## Dynamic Type Conversion

The TYPE keyword to the FIX function allows type conversion to an arbitrary type at runtime without the use of CASE or IF statements on each type. The following example demonstrates the use of the TYPE keyword:

```

PRO EXAMPLE_FIXTYPE
  ; Define a variable as a double:
  A = 3D
  ; Store the type of A in a variable:
  typeA = SIZE(A, /TYPE)
  PRINT, 'A is type code', typeA
  ; Prompt the user for a numeric value:
  READ, UserVal, PROMPT='Enter any Numeric Value: '
  ; Convert the user value to the type stored in typeA:
  ConvUserVal = FIX(UserVal, TYPE=typeA)
  PRINT, ConvUserVal
END

```



IDL Type Conversion functions are described further in [Chapter 5, “Constants”](#).

# Arrays

*Arrays* are multidimensional data sets which are manipulated according to mathematical rules. Array elements can be of any IDL data type, but all elements of a given array must be of the same data type. Array *subscripts* provide a means of selecting one or more elements of an array for retrieval or modification.

This section provides a brief overview of IDL arrays. For more detailed information, see [Chapter 8, “Array Subscripts”](#).

One-dimensional arrays are often called *vectors*. The following IDL statement creates a vector with five single-precision floating-point elements:

```
array = [1.0, 2.0, 3.0, 4.0, 5.0]
```

Two-dimensional arrays are often used in image processing and in mathematical operations (where they are often termed *matrices*). The following IDL statement creates a three-column by two-row array:

```
array = [[1, 2, 3], [4, 5, 6]]
```

Use the PRINT procedure to display the contents of the array:

```
PRINT, array
```

IDL prints:

```

1      2      3
4      5      6
```

Arrays can have up to eight dimensions in IDL. The following IDL statement creates a three-column by four-row by five-layer deep three-dimensional array. In this case, we use the IDL FINDGEN function to create an array whose elements are set equal to the floating-point values of their one-dimensional subscripts:

```
array = FINDGEN(3, 4, 5)
```

IDL is an array-oriented language. This means that array operations execute more efficiently than similar one-dimensional operations. For example, suppose you have a three-dimensional array and wish to divide each element by two. A language that does not support array operations would create a loop to perform the division for each element; IDL accomplishes the division in a single line of code:

```
array = array/2
```

## Array Subscripts

Individual array elements can be referenced using their *subscripts*. In IDL, array subscripts are *zero-based*; this means that the first element in an array is element zero, the second is element one, etc. For example, in the array created by the following IDL statement:

```
array = [ 1, 2, 3 ]
```

The integer 1 is element zero of the array, the integer 2 is element one, and the integer 3 is element two. The following IDL statement creates a new variable that contains element one of array:

```
new = array[1]
```

Displaying the value of `new` reveals the following:

```
PRINT, new
```

IDL prints:

```
2
```

The values of the selected array elements are extracted when a subscripted variable reference appears in an expression. New values are stored in selected array elements, without disturbing the remaining elements, when a subscript reference appears on the left side of an assignment statement. See “[The Assignment Statement](#)” on page 198 for information on the use of the different types of assignment statements when storing into arrays.

The syntax of a subscript reference is:

*Variable\_Name* [*Subscript\_List*]

or

(*Array\_Expression*)[*Subscript\_List*]

The *Subscript\_List* is simply a list of expressions, constants, or subscript ranges containing the values of one or more subscripts. Subscript expressions are separated by commas if there is more than one subscript. In addition, multiple elements are selected with subscript expressions that contain either a contiguous range of subscripts or an array of subscripts.

### Note

In versions of IDL prior to version 5.0, parentheses were used to enclose array subscripts instead of square brackets. While the old syntax (using parentheses to enclose array subscripts) will continue to work, we suggest that you use square

brackets in all new code. See [“Array Subscript Syntax: \[ \] vs. \( \)”](#) on page 157 for a discussion of the change.

---

# Structures

IDL supports structures and arrays of structures. A structure is a collection of scalars, arrays, or other structures contained in a variable. Structures are useful for representing data in a natural form, transferring data to and from other programs, and containing a group of related items of various types. There are two types of structures: named and anonymous. Named structures are used when the definitions will not be changed, since the attributes of the structure are stored internally in IDL and cannot be changed in the current IDL session. Anonymous structures are useful when the structure, type, and/or dimensions of its components change during program execution.

This section provides a brief overview of IDL structure variables. For more detailed information, see [Chapter 7, “Structures”](#).

## Creating and Defining Structures

A named structure is created by executing a structure-definition expression, which is an expression of the following form:

```
{Structure_Name, Tag_Name1 : Tag_Definition1, ..., Tag_Namen : Tag_Definitionn}
```

Anonymous structures are created in the same way, but with the structure’s name omitted.

```
{Tag_Name1 : Tag_Definition1, ..., Tag_Namen : Tag_Definitionn}
```

Anonymous structures can also be created and combined using the `CREATE_STRUCT` function.

For example, assume that a star catalog is to be processed. Each entry for a star contains the following information: star name, right ascension, declination, and an intensity measured each month over the last 12 months. A structure for this information is defined with the following IDL statement:

```
A = {STAR, NAME: '', RA: 0.0, DEC: 0.0, INTEN: FLTARR(12)}
```

Each tag name, NAME, RA, DEC and INTEN, is followed by its tag definition.

The same structure is created as an anonymous structure by the statement:

```
A = {NAME: '', RA: 0.0, DEC: 0.0, INTEN: FLTARR(12)}
```

or by using the `CREATE_STRUCT` function:

```
A = CREATE_STRUCT('NAME', '', 'RA', 0.0, 'DEC', 0.0, $
                  'INTEN', FLTARR(12))
```

## Structure References

The basic syntax of a reference to a field within a structure is as follows:

*Variable\_Name.Tag\_Name*

### Examples of Structure References

The name of the star contained in A is referenced as A.NAME. The entire intensity array is referred to as A.INTEN, while the *n*th element of A.INTEN is A.INTEN[N]. The following are valid IDL statements using the STAR structure:

```

;Store a structure of type STAR into variable A. Define the values
;of all fields.
A = {STAR, NAME: 'SIRIUS', RA: 30., DEC: 40., INTEN:FINDGEN(12)}

;Set name field only.
A.NAME = 'BETELGEUSE'

;Print name, right ascension, and declination.
PRINT, A.NAME, A.RA, A.DEC

;Set Q to the value of the sixth element of A.INTEN. Q will be a
;floating-point scalar.
Q = A.INTEN[5]

;Set RA field to 23.21.
A.RA = 23.21

;Zero all 12 elements of intensity field.
A.INTEN = 0

;Store 4th through 7th elements of INTEN field in variable B.
B = A.INTEN[3:6]

;The integer 12 is converted to string and stored in the name field
;because the field is defined as a string.
A.NAME = 12

;Copy A to B. The entire structure is copied and B contains a STAR
;structure.
B = A

```

## Using HELP with Structures

The statement

```
HELP, /STRUCTURE, A
```

shows the type, structure and tag name of each field in a structure.

## Parameter Passing with Structures

An *entire* structure is passed by *reference* by simply using the name of the variable containing the structure as a parameter. For example, the following statement prints the value of the structure field `A.NAME` :

```
PRINT, A.NAME
```

# Variables

Variables are named repositories where information is stored. A variable can have virtually any size and can contain any of the IDL data types. Variables can be used to store images, spectra, single quantities, names, tables, etc.

This section provides a brief overview of IDL variables. For more detailed information, see [Chapter 5, “Constants”](#).

## Attributes of Variables

Every variable has a number of attributes that can change during the execution of a program or terminal session. Variables have both a *structure* and a *type*.

### Structure

A variable can contain a single value (a scalar) or a number of values of the same type (an array) or data entities of potentially differing type and size (a structure). Strings are considered as single values, and a string array contains a number of variable-length strings.

In addition, a variable can associate an array structure with a file; these variables are called associated variables. Referencing an associated variable causes data to be read from, or written to, the file. Associated variables are described in [ASSOC](#) in the *IDL Reference Guide*.

### Type

A variable can have one and only one of the following types: undefined, byte, integer, unsigned integer, 32-bit longword, unsigned 32-bit longword, 64-bit integer, unsigned 64-bit integer, floating-point, double-precision floating-point, complex floating-point, double-precision complex floating-point, string, structure, pointer, or object reference.

When a variable appears on the left-hand side of an assignment statement, its attributes are copied from those of the expression on the right-hand side. For example, the statement

```
ABC = DEF
```

redefines or initializes the variable ABC with the attributes and value of variable DEF. Attributes previously assigned to the variable are destroyed. Initially, every variable has the single attribute of undefined. Attempts to use the value of an undefined variables result in an error.



## Variable Names

IDL variables are named by identifiers. Each identifier must begin with a letter and can contain from 1 to 128 characters. The second and subsequent characters can be letters, digits, the underscore character, or the dollar sign. A variable name cannot contain embedded spaces, because spaces are considered to be delimiters. Characters after the first 128 are ignored. Names are case insensitive. Lowercase letters are converted to uppercase; so the variable name `abc` is equivalent to the name `ABC`. The following table illustrates some acceptable and unacceptable variable names.

Unacceptable	Reason	Acceptable
EOF	Conflicts with function name	A
6A	Does not start with letter	A6
_INIT	Does not start with letter	INIT_STATE
AB@	Illegal character	ABC\$DEF
ab cd	Embedded space	My_variable

*Table 4-5: Unacceptable and Acceptable IDL Variable Names*

### Warning

---

A variable cannot have the same name as a function (either built-in or user defined) or a reserved word (see the following list). Giving a variable such a name results in a syntax error or in “hiding” the variable.

---

The following table lists all of the reserved words in IDL.

AND	BEGIN	CASE	COMMON
DO	ELSE	END	ENDCASE
ENDELSE	ENDFOR	ENDIF	ENDREP
ENDWHILE	EQ	FOR	FUNCTION
GE	GOTO	GT	IF
LE	LT	MOD	NE
NOT	OF	ON_IOERROR	OR
PRO	REPEAT	THEN	UNTIL
WHILE	XOR		

# System Variables

System variables are a special class of predefined variables available to all program units. Their names always begin with the exclamation mark character (!). System variables are used to set the options for plotting, to set various internal modes, to return error status, etc.

System variables have a predefined type and structure that cannot be changed. When an expression is stored into a system variable, it is converted to the variable type, if necessary and possible. Certain system variables are *read only*, and their values cannot be changed. The user can define new system variables with the `DEFSYSV` procedure.

System variables are discussed in [Appendix D, “System Variables”](#) in the *IDL Reference Guide*.





# Chapter 5: Constants

The following topics are covered in this chapter:

---

<a href="#">Data Types</a> .....	102	<a href="#">Type Conversion Functions</a> .....	110
<a href="#">Constants</a> .....	104		

# Data Types

The IDL language is *dynamically typed*. This means that an operation on a variable can change that variable's type. In general, when variables of different types are combined in an expression, the result has the data type that yields the highest precision.

For example, if an integer variable is added to a floating-point variable, the result will be a floating-point variable.

## Basic Data Types

In IDL there are twelve basic, atomic data types, each with its own form of constant. The data type assigned to a variable is determined either by the syntax used when creating the variable, or as a result of some operation that changes the type of the variable.

IDL's basic data types are discussed in more detail beginning with [“Constants”](#) on page 104.

- **Byte:** An 8-bit unsigned integer ranging in value from 0 to 255. Pixels in images are commonly represented as byte data.
- **Integer:** A 16-bit signed integer ranging from  $-32,768$  to  $+32,767$ .
- **Unsigned Integer:** A 16-bit unsigned integer ranging from 0 to 65535.
- **Long:** A 32-bit signed integer ranging in value from approximately minus two billion to plus two billion.
- **Unsigned Long:** A 32-bit unsigned integer ranging in value from 0 to approximately four billion.
- **64-bit Long:** A 64-bit signed integer ranging in value from  $-9,223,372,036,854,775,808$  to  $+9,223,372,036,854,775,807$ .
- **64-bit Unsigned Long:** A 64-bit unsigned integer ranging in value from 0 to 18,446,744,073,709,551,615.
- **Floating-point:** A 32-bit, single-precision, floating-point number in the range of  $\pm 10^{38}$ , with approximately six or seven decimal places of significance.
- **Double-precision:** A 64-bit, double-precision, floating-point number in the range of  $\pm 10^{308}$  with approximately 14 decimal places of significance.

- **Complex:** A real-imaginary pair of single-precision, floating-point numbers. Complex numbers are useful for signal processing and frequency domain filtering.
- **Double-precision complex:** A real-imaginary pair of double-precision, floating-point numbers.

---

**Note**

In previous versions of IDL prior to version 4, the combination of a double-precision number and a complex number in an expression resulted in a single-precision complex number because those versions of IDL lacked the DCOMPLEX double-precision complex data type. Starting with IDL version 4, this combination results in a DCOMPLEX number.

---

- **String:** A sequence of characters, from 0 to 32,767 characters in length, which is interpreted as text.

### Precision of Floating-Point Numbers

The precision of IDL's floating-point numbers depends somewhat on the platform involved and the compiler and specific compiler switches used to compile the IDL executable. The values shown here are minimum values; in some cases, IDL may deliver slightly more precision than we have indicated. If your application uses numbers that are sensitive to floating-point truncation or round-off errors, or values that cannot be represented exactly as floating-point numbers, this is something you should consider.

For more information on floating-point mathematics, see [Chapter 16, "Mathematics"](#) in the *Using IDL* manual. For information on your machine's precision, see [MACHAR](#) in the *IDL Reference Guide*.

### Complex Data Types

- **Structures:** Aggregations of data of various types. Structures are discussed in [Chapter 7, "Structures"](#).
- **Pointers:** A reference to a dynamically-allocated *heap variable*. Pointers are discussed in [Chapter 11, "Pointers"](#).
- **Object References:** A reference to a special heap variable that contains an IDL object structure. Object references are discussed in [Chapter 12, "Object Basics"](#).

# Constants

## Integer Constants

Numeric constants of different types can be represented by a variety of forms. The syntax used when creating integer constants is shown in the following table, where *n* represents one or more digits.

Radix	Type	Form	Examples
Decimal	Byte	<i>n</i> B	12B, 34B
	Integer	<i>n</i> or <i>n</i> S	12,12S,425,425S
	Unsigned Integer	<i>n</i> U or <i>n</i> US	12U,12US
	Long	<i>n</i> L	12L, 94L
	Unsigned Long	<i>n</i> UL	12UL, 94UL
	64-bit Long	<i>n</i> LL	12LL, 94LL
	Unsigned 64-bit Long	<i>n</i> ULL	12ULL, 94ULL
Hexadecimal	Byte	' <i>n</i> 'XB	'2E'XB
	Integer	' <i>n</i> 'X	'0F'X
	Unsigned Integer	' <i>n</i> 'XU	'0F'XU
	Long	' <i>n</i> 'XL	'FF'XL
	Unsigned Long	' <i>n</i> 'XUL	'FF'XUL
	64-bit Integer	' <i>n</i> 'XLL	'FF'XLL
	Unsigned 64-bit Integer	' <i>n</i> 'XULL	'FF'XULL

Table 5-1: Integer Constants



Radix	Type	Form	Examples
Octal	Byte	"nB	"12B
	Integer	"n	"12
		'n'O	'377'O
	Unsigned Integer	"nU	"12U
		'n'OU	'377'OU
	Long	"nL	"12L
		'n'OL	'777777'OL
	Unsigned Long	"nUL	"12UL
		'n'OUL	'777777'OUL
	64-bit Long	"nLL	"12LL
		'n'OLL	'777777'OLL
	Unsigned 64-bit	"nULL	"12ULL
	Long	'n'OULL	'777777'OULL

*Table 5-1: Integer Constants*

Digits in hexadecimal constants include the letters A through F for the decimal numbers 10 through 15. Octal constant use the same style as hexadecimal constants, substituting an O for the X.

Absolute values of integer constants are given in the following table.

Type	Absolute Value Range
Byte	0 – 255
Integer	0 – 32767
Unsigned Integer	0 – 65535
Long	0 – $2^{31} - 1$
Unsigned Long	0 – $2^{32} - 1$
64-bit Long	0 – $2^{63} - 1$

*Table 5-2: Absolute Value Range Of Integer Constants*

Type	Absolute Value Range
Unsigned 64-bit Long	$0 - 2^{64} - 1$

*Table 5-2: Absolute Value Range Of Integer Constants*

Integers specified without one of the B, S, L, or LL specifiers are automatically promoted to an integer type capable of holding them. For example, 40000 is promoted to longword because it is too large to fit in an integer. Any numeric constant can be preceded by a plus (+) or minus (-) sign. The following table illustrates examples of both valid and invalid IDL constants.

Unacceptable	Reason	Acceptable
256B	Too large, limit is 255	255B
'123L	Missing apostrophe	'123'L
'03G'x	Invalid character	"129
'27'L	No radix	'27'OL
650XL	No apostrophes	'650'XL
"129	9 is an invalid octal digit	"124

*Table 5-3: Examples of Integer Constants*

## Floating-Point and Double-Precision Constants

Floating-point and double-precision constants can be expressed in either conventional or scientific notation. Any numeric constant that includes a decimal point is a floating-point or double-precision constant.

The syntax of floating-point and double-precision constants is shown in the following table. The notation “*sx*” represents the sign and magnitude of the exponent, for example, E-2.

Form	Example
<i>n.</i>	102.
<i>.n</i>	.102
<i>n.n</i>	10.2
<i>nE<sub>sx</sub></i>	10E5
<i>n.E<sub>sx</sub></i>	10.E-3
<i>.nE<sub>sx</sub></i>	.1E+12
<i>n.nE<sub>sx</sub></i>	2.3E12

Table 5-4: Syntax of Floating-Point Constants

Double-precision constants are entered in the same manner, replacing the E with a D. For example, 1.0D0, 1D, and 1.D each represent a double-precision numeral 1.

## Complex Constants

Complex constants contain a real and an imaginary part, both of which are single- or double-precision floating-point numbers. The imaginary part can be omitted, in which case it is assumed to be zero. The form of a complex constant is as follows:

```
COMPLEX(REAL_PART, IMAGINARY_PART)
```

or

```
COMPLEX(REAL_PART)
```

For example, COMPLEX(1,2) is a complex constant with a real part of one, and an imaginary part of two. COMPLEX(1) is a complex constant with a real part of one and a zero imaginary component. To extract the real part of a complex expression, use the REAL function. The ABS function returns the magnitude of a complex expression, and the IMAGINARY function returns the imaginary part.

## String Constants

A string constant consists of zero or more characters enclosed by apostrophes (') or quotes ("). The value of the constant is simply the characters appearing between the

leading delimiter ( ' or "" ) and the next occurrence of the *same* delimiter. A double apostrophe ( ' ' ) or quote ( " " ) is considered to be the null string; a string containing no characters. An apostrophe or quote can be represented within a string by two apostrophes or quotes; e.g., 'Don't' returns Don't. This syntax often can be avoided by using a different delimiter; e.g., "Don't" instead of 'Don't'. The following table illustrates valid string constants.

Expression	Resulting String
'Hi there'	Hi there
"Hi there"	Hi there
' '	Null String
"I'm happy"	I'm happy
'I'm happy'	I'm happy
'counter'	counter
'129'	129

*Table 5-5: Examples of Valid String Constants*

The following table illustrates invalid string constants. In the last entry of the table, "129" is interpreted as an illegal octal constant. This is because a quote character followed by a digit from 0 to 7 represents an octal numeric constant, not a string, and the character 9 is an illegal octal digit.

String Value	Unacceptable	Reason
Hi there	'Hi there"	Mismatched delimiters
Null String	'	Missing delimiter
I'm happy	'I'm happy'	Apostrophe in string
counter	"counter"	Double apostrophe is null string
129	"129"	Illegal octal constant

*Table 5-6: Examples of Invalid String Constants*

**Note**

While an IDL string variable can hold up to 64 Kbytes of information, the buffer that handles input at the IDL command prompt is limited to 255 characters. If for some reason you need to create a string variable longer than 255 characters at the IDL command prompt, split the variable into multiple sub-variables and combine them with the “+” operator:

```
var = var1+var2+var3
```

This limit only affects string constants created at the IDL command prompt.

## Representing Non-Printable Characters

The ASCII characters with value less than 32 or greater than 126 do not have printable representations. Such characters can be included in string constants by specifying their ASCII value as a byte argument to the `STRING` function. The following table gives examples of using octal or hexadecimal character notation.

Specified String	Actual Contents	Comment
<code>STRING(27B)+'[;H' +STRING(27B)+[2J'</code>	'<Esc>[;H<Esc>[2J'	Erase ANSI terminal
<code>STRING(7B)</code>	Bell	Ring the bell
<code>STRING(8B)</code>	Backspace	Move cursor left

*Table 5-7: Specifying Non-Printable Characters*

Note that ASCII characters may have different effects (or no effect) on platforms that do not support ASCII terminal commands.

# Type Conversion Functions

IDL allows you to convert data from one data type to another using a set of conversion functions. These functions are useful when you need to force the evaluation of an expression to a certain type, output data in a mode compatible with other programs, etc. The conversion functions are in the following table:

Function	Description
STRING	Convert to string
BYTE	Convert to byte
FIX	Convert to 16-bit integer, or optionally other type
UINT	Convert to 16-bit unsigned integer
LONG	Convert to 32-bit integer
ULONG	Convert to 32-bit unsigned integer
LONG64	Convert to 64-bit integer
ULONG64	Convert to 64-bit unsigned integer
FLOAT	Convert to floating-point
DOUBLE	Convert to double-precision floating-point
COMPLEX	Convert to complex value
DCOMPLEX	Convert to double-precision complex value

*Table 5-8: Type Conversion Functions*

Conversion functions operate on data of any structure: scalars, vectors, or arrays, and variables can be of any type.

## Take Care When Converting Types

If the variable you are converting lies outside the range of the type to which you are converting, IDL will truncate the binary representation of the value without informing you. For example:

```

; Define A. Note that the value of A is outside the range
; of integers, and is automatically created as a longword
; integer by IDL.
A = 33000

```

```

;B is silently truncated.
B = FIX(A)
PRINT, B

```

IDL prints:

```
-32536
```

Applying `FIX` creates a short (16-bit) integer. If the value of the variable passed to `FIX` lies outside the range of 16-bit integers, IDL will silently truncate the binary value, returning only the 16 least-significant bits, with no indication that an error has occurred.

With most floating-point operations, error conditions can be monitored using the `FINITE` and `CHECK_MATH` functions. See [Chapter 15, “Controlling Errors”](#), for more information.

## Converting Strings

When converting from a string argument, it is possible that the string does not contain a valid number and no conversion is possible. The default action in such cases is to print a warning message and return zero. The `ON_IOERROR` procedure can be used to establish a statement to be jumped to in case of such errors.

Conversion between strings and byte arrays (or vice versa) is something of a special case. The result of the `BYTE` function applied to a string or string array is a byte array containing the ASCII codes of the characters of the string. Converting a byte array with the `STRING` function yields a string array or scalar with one less dimension than the byte array.

## Dynamic Type Conversion

The `TYPE` keyword to the `FIX` function allows type conversion to an arbitrary type at runtime without the use of `CASE` or `IF` statements on each type. The following example demonstrates the use of the `TYPE` keyword:

```

PRO EXAMPLE_FIXTYPE
; Define a variable as a double:
A = 3D

; Store the type of A in a variable:
typeA = SIZE(A, /TYPE)
PRINT, 'A is type code', typeA

; Prompt the user for a numeric value:
READ, UserVal, PROMPT='Enter any Numeric Value: '

```

```

; Convert the user value to the type stored in typeA:
ConvUserVal = FIX(UserVal, TYPE=typeA)

PRINT, ConvUserVal
END

```

## Examples of Type Conversion

See the following table for examples of type conversions and their results.

Operation	Results
FLOAT(1)	1.0
FIX(1.3 + 1.7)	3
FIX(1.3) + FIX(1.7)	2
FIX(1.3, TYPE=5)	1.3000000
BYTE(1.2)	1
BYTE(-1)	255b (Bytes are modulo 256)
BYTE('01ABC')	[48b, 49b, 65b, 66b, 67b]
STRING([65B, 66B, 67B])	'ABC'
FLOAT(COMPLEX(1, 2))	1.0
COMPLEX([1, 2], [4, 5])	[COMPLEX(1,4),COMPLEX(2,5)]

*Table 5-9: Uses of Type Conversion Functions*





## Chapter 6:

# Expressions and Operators

The following topics are covered in this chapter:

---

Overview .....	114	IDL Operators .....	117
Operator Precedence .....	115	Type and Structure of Expressions .....	127

## Overview

Variables and constants are combined into *expressions* using operators and functions, and providing a result. Expressions can be combined with other expressions, variables, and constants to yield more complex expressions. In IDL, unlike FORTRAN or C, expressions can be scalar- or array-valued.

There are many types of operators in IDL. In addition to the usual operators — addition, subtraction, multiplication, division, exponentiation, relations (EQ, NE, GT, etc.), and Boolean arithmetic (AND, OR, NOT, and XOR) — other operators exist to find minima, maxima, select scalars and subarrays from arrays (subscripting), and to concatenate scalars and arrays to form new arrays.

Functions, which are operators in themselves, perform operations that are usually of a more complex nature than those denoted by simple operators. Functions exist in IDL for data smoothing, shifting, transforming, evaluation of transcendental functions, and other operations.

Expressions can be arguments to functions or procedures. For example, the expression `SIN(A*!PI)` evaluates the variable `A` multiplied by the value of  $\pi$ , then applies the trigonometric sine function. This result can be used as an operand to form a more complex expression or as an argument to yet another function (e.g., `EXP(SIN(A*!PI))` evaluates  $e^{\sin \pi a}$ ).

# Operator Precedence

IDL operators are divided into the levels of algebraic precedence found in common arithmetic. Operators with higher precedence are evaluated before those with lesser precedence, and operators of equal precedence are evaluated from left to right. Operators are grouped into five classes of precedence as shown in the following table.

Priority	Operator
First (highest)	( ) (parentheses, to group expressions)
Second	* (pointer dereference)
	^ (exponentiation)
Third	* (multiplication)
	# and ## (matrix multiplication)
	/(division)
	MOD (modulus)
Fourth	+ (addition)
	- (subtraction and negation)
	< (minimum)
	> (maximum)
	NOT (Boolean negation)
Fifth	EQ (equality)
	NE (not equal)
	LE (less than or equal)
	LT (less than)
	GE (greater than or equal)
	GT (greater than)

Table 6-1: Operator Precedence

Priority	Operator
Sixth	AND (Boolean AND)
	OR (Boolean OR)
	XOR (Boolean exclusive OR)
Seventh	? (conditional expression)

*Table 6-1: Operator Precedence*

The effect of operators is based on precedence, not position. This concept is shown by the following examples.

$$A = 4 + 5 * 2$$

A is equal to 14 since the multiplication operator has a higher precedence than the addition operator. Parentheses can be used to override the default evaluation.

$$A = (4 + 5) * 2$$

In this case, A equals 18 because the expression inside the parentheses is evaluated first.

A useful rule of thumb is, “when in doubt, parenthesize”. Some examples of expressions are provided in the following table.

Expression	Value
$A + 1$	The sum of A and 1.
$A < 2 + 1$	The smaller of A or two, plus one.
$A < 2 * 3$	The smaller of A and six, since * has higher precedence than <.
$2 * \text{SQRT}(A)$	Twice the square root of A.
$A + \text{'Thursday'}$	The concatenation of the strings A and “Thursday.” An error results if A is not a string

*Table 6-2: Examples of Expressions*

# IDL Operators

As described above, operators are used to combine terms and expressions. IDL supports the following types of operators:

- [Parentheses](#)
- [Square Brackets](#)
- [Mathematical Operators](#)
- [Minimum and Maximum Operators](#)
- [Matrix Multiplication](#)
- [Array Concatenation](#)
- [Boolean Operators](#)
- [Relational Operators](#)

## Parentheses

Parentheses are used to group expressions and to enclose function parameter lists. Parentheses can be used to override the order of operator evaluation described above. Examples:

```
;Parentheses enclose function argument lists.
SIN(ANG * PI/180.)

;Parentheses specify order of operator evaluation.
(A + 5)/B
```

The right parenthesis must always close the list begun by the left parenthesis.

## Square Brackets

Square brackets are used to create arrays and to enclose array subscripts.

```
;Use brackets when assigning elements to an array.
ARRAY = [1, 2, 3, 4, 5]

;Brackets enclose subscripts.
ARRAY[X, Y]
```

### Note

In versions of IDL prior to version 5.0, parentheses were used to enclose array subscripts. While using parentheses to enclose array subscripts will continue to

work as in previous version of IDL, we strongly suggest that you use brackets in all new code. See “[Array Subscript Syntax: \[ \] vs. \( \)](#)” on page 157 for additional details.

---

## Mathematical Operators

There are seven basic IDL mathematical operators, described below.

### Assignment

The equal sign (=) is the assignment operator. The value of the expression on the right hand side of the equal sign is stored in the variable, subscript element, or range on the left side. See “[The Assignment Statement](#)” on page 198. For example, the following assigns the value 32 to A.

```
A = 32
```

### Addition

The positive sign (+) is the addition operator. When applied to strings, the addition operator concatenates the strings. For example:

```
;Store the sum of 3 and 6 in B.  
B = 3 + 6
```

```
;Store the string value of "John Doe" in B.  
B = 'John' + ' ' + 'Doe'
```

### Subtraction and Negation

The negative sign (-) is the subtraction operator. Also, the minus sign is used as the unary negation operator. For example:

```
;Store the value of 5 subtracted from 9 in C.  
C = 9 - 5
```

```
;Change the sign of C.  
C = -C
```

### Multiplication

The asterisk (\*) is the multiplication operator. For example, Store the product of 2 and 5 in variable C:

```
C = 2 * 5
```

## Division

The forward slash (/) is the division operator. For example, Store the result of 10.0 divided by 3.2 in variable D:

```
D = 10.0/3.2
```

## Exponentiation

The caret (^) is the exponentiation operator.  $A^B$  is equal to A raised to the B power.

- If A is a real number and B is of integer type, repeated multiplication is applied.
- If A is real and B is real (non-integer), the formula  $A^B = e^{B \ln A}$  is evaluated.
- If A is complex and B is real, the formula  $A^B = (re^{i\theta})^B = r^B (\cos B\theta + i \sin B\theta)$  (where  $r$  is the real part of A and  $i\theta$  is the imaginary part) is evaluated.
- If B is complex, the formula  $A^B = e^{B \ln A}$  is evaluated. If A is also complex, the natural logarithm is computed to be  $\ln(A) = \ln(re^{i\theta}) = \ln(r) + i\theta$  (where  $r$  is the real part of A and  $i\theta$  is the imaginary part).
- $A^0$  is defined as 1.

## Modulo

The keyword MOD is the modulo operator.  $I \text{ MOD } J$  is equal to the remainder when I is divided by J. The magnitude of the result is less than that of J, and its sign agrees with that of I. For example:

```
;Assign the value of 9 modulo 5 (4) to A.
A = 9 MOD 5

;Compute angle modulo 2p.
A =(ANGLE + B) MOD (2 * !PI)
```

## Minimum and Maximum Operators

The IDL minimum and maximum operators return the smaller or larger of their operands, as described below. Note that negated values must be enclosed in parentheses in order for IDL to interpret them correctly.

### The Minimum Operator

The “less than” sign (<) is the IDL minimum operator. The value of “ $A < B$ ” is equal to the smaller of A or B. For example:

```
;Set A equal to 3.
```

```

A = 5 < 3

;Set A equal to -6.
A = 5 < (-6)

;Syntax Error. IDL attempts to perform a subtraction operation if
;the "-6" is not enclosed in parentheses.
A = 5 < -6

;Set all points in array ARR that are larger than 100 to 100.
ARR = ARR < 100

;Set X to the smallest of the three operands.
X = X0 < X1 < X2

```

## The Maximum Operator

The “greater than” sign (>) is the IDL maximum operator. “A > B” is equal to the larger of A or B. For example:

```

;'>' is used to avoid taking the log of zero or negative numbers.
C = ALOG(D > 1E - 6)

;Plot positive points only. Negative points are plotted as zero.
PLOT, ARR > 0

```

## Matrix Multiplication

IDL has two operators used to multiply arrays and matrices.

### The # Operator

The # operator computes array elements by multiplying the columns of the first array by the rows of the second array. The second array must have the same number of columns as the first array has rows. The resulting array has the same number of columns as the first array and the same number of rows as the second array.

### The ## Operator

The ## operator does what is commonly referred to as *matrix multiplication*. It computes array elements by multiplying the rows of the first array by the columns of the second array. The second array must have the same number of rows as the first array has columns. The resulting array has the same number of rows as the first array and the same number of columns as the second array.

For an example illustrating the difference between the two, see [Multiplying Arrays](#) in the *Using IDL* manual.



## Array Concatenation

The square brackets are used as array concatenation operators. Operands enclosed in square brackets and separated by commas are concatenated to form larger arrays. The expression `[A,B]` is an array formed by concatenating A and B, which can be scalars or arrays, along the first dimension.

Similarly, `[A,B,C]` concatenates A, B, and C. The second and third dimensions can be concatenated by nesting the bracket levels; `[[1,2],[3,4]]` is a 2-element by 2-element array with the first row containing 1 and 2 and the second row containing 3 and 4. Operands must have compatible dimensions; all dimensions must be equal except the dimension that is to be concatenated, e.g., `[2,INTARR(2,2)]` are incompatible. Examples:

```
;Define C as three-point vector.
C = [-1, 1, -1]

;Add 12 to the end of C.
C = [C, 12]

;Insert 12 at the beginning of C.
C = [12, C]

;Plot ARR2 appended to ARR1.
PLOT, [ARR1, ARR2]

;Define a 3x3 matrix.
KER = [[1,2,1], [2,4,2], [1,2,1]]
```

### Note

The maximum number of operands that can appear within brackets varies across IDL implementations but is always at least 25. If you must create an array of more than 25 elements using the concatenation operator, use multiple statements. For example, to create an array with 70-constant elements, use the following statements:

```
A = [k0, k1, ..., k24]
A = [A, k25, k26, ..., k49]
A = [A, k50, k51, ..., k69]
```

This method is relatively inefficient and should be performed only once if possible.

## Boolean Operators

There are four Boolean operators in IDL. Boolean operators return either “true” or “false” as described previously. Note that the Boolean operators do not work with string and complex arguments.

### AND

AND is a Boolean operator that returns “true” whenever both of its operands are true; otherwise, the result is “false.” Any nonzero value is considered true. For integer, longword, and byte operands, a bitwise AND operation is performed. For operations on other types, the result is equal to the second operand if the first operand is not equal to zero or the null string; otherwise, the result is zero or the null string.

### NOT

The NOT operator is the Boolean inverse and is a unary operator (it has only one operand). In other words, “NOT true” is equal to “false” and “NOT false” is equal to “true.” NOT complements each bit for integer operands.

### Note

---

Signed integers are expressed using the “2s complement” representation. This means that to arrive at the decimal representation of a negative binary number (a string of binary digits with a one as the most significant bit), you must take the complement of each bit, add one, convert to decimal, and prepend a negative sign. This means that NOT 0 equals -1, NOT 1 equals -2, etc.

---

For floating-point operands, the result is 1.0 if the operand is zero; otherwise, the result is zero. The NOT operator is not valid for string or complex operands.

### OR

OR is the Boolean inclusive operator. For integer or byte operands, a bitwise inclusive OR is performed. For example, 3 OR 5 equals 7. For floating-point operands, the OR operator returns the first operand if it is non-zero, or the 2nd operand otherwise.

### XOR

XOR is the Boolean “exclusive or” function. XOR is only valid for byte, integer, and longword operands. A bit in the result is set to 1 if the corresponding bits in the operands are different; if they are equal, it is set to zero.

The following table summarizes the action of the boolean operators:

<b>Operator(<i>op</i>)</b>	<b>T <i>op</i> T</b>	<b>T <i>op</i> F</b>	<b>F <i>op</i> F</b>
AND	T	F	F
OR	T	T	F
XOR	F	T	F
	<b><i>op</i> T</b>	<b><i>op</i> F</b>	
NOT	F	T	

*Table 6-3: Action of Boolean Operators*

When applied to bytes, integers, and longword operands, the Boolean functions operate on each binary bit. For example:

<b>Decimal</b>	<b>Binary</b>
3 AND 5 = 1	0011 AND 0101 = 0001
3 OR 5 = 7	0011 OR 0101 = 0111
3 XOR 5 = 6	0011 XOR 0101 = 0110
NOT 5 = -6	NOT 0101 = 1010

*Table 6-4: Action of Boolean Operators on Integers*

Results of relational expressions can be combined into more complex expressions using the Boolean operators. Some examples of relational and Boolean expressions are as follows:

```
;True if A is between 25 and 50. If A is an array, then the result
;is an array of zeros and ones.
(A LE 50) AND (A GE 25)
```

```
;True if A is less than 25 or greater than 50. This is the inverse
;of the first.
(A GT 50) OR (A LT 25)
```

```
;Adds (using the logical AND operator) the hexadecimal constant FF
;(255 in decimal) to the array ARR. This masks the lower 8-bits and
;zeros the upper bits.
ARR AND 'FF'X
```

## Relational Operators

The IDL relational operators can be used to test the relationship between two arguments. The six relational operators are described in the following table:

Operator	Description
EQ	Equal to
NE	Not equal to
GE	Greater than or equal to
GT	Greater than
LE	Less than or equal to
LT	Less than

*Table 6-5: Relational Operators*

Relational operators apply a relation to two operands and return a logical value of true or false. The resulting logical value can be used as the predicate in IF, WHILE or REPEAT statements can be combined using Boolean operators with other logical values to make more complex expressions. For example: “1 EQ 1” is true, and “1 GT 3” is false.

The rules for evaluating relational expressions with operands of mixed modes are the same as those given above for arithmetic expressions. For example, in the relational expression “2 EQ 2.0”, the integer 2 is converted to floating point and compared to the floating point 2.0. The result of this expression is true, as represented by a byte 1.

In IDL, the value “true” is represented by the following:

- Any odd, nonzero value for byte, integer, and longword data types
- Any nonzero value for single, double-precision, and the real part of a complex number (the imaginary part is ignored)
- Any non-null string

Conversely, false is represented as anything that is not true—zero or even-valued integers; zero-valued, floating-point quantities; and the null string.

The relational operators return a value of 1 for true and 0 for false. The type of the result is always byte.

**EQ**

EQ is the relational “equal to” operator. This operator returns true if its operands are equal; otherwise, it returns false. This operator always returns a byte value of 1 for true and a byte value of 0 for false.

**NE**

NE is the “not-equal-to” relational operator. This operator returns true whenever the operands are different. For example "sun" NE "fun" returns true.

**GE**

GE is the “greater than or equal to” relational operator. The GE operator returns true if the operand on the left is greater than or equal to the one on the right. One use of relational operators is to mask arrays as shown in the following statement:

```
A = ARRAY * (ARRAY GE 100)
```

This command sets A equal to ARRAY whenever the corresponding element of ARRAY is greater than or equal to 100. If the element is less than 100, the corresponding element of A is set to zero.

Strings are compared using the ASCII collating sequence: " " is less than "0" is less than "9" is less than "A" is less than "Z" is less than "a" which is less than "z".

**GT**

GT is the “greater than” relational operator. This operator returns true if the operand on the left is greater than the operand on the right. For example, “6 GT 5” returns true.

**LE**

LE is the “less-than or equal-to” relational operator. This operator returns true if the operand on the left is less than or equal to the operand on the right. For example, “4 LE 4” returns true.

**LT**

LT is the “less-than” relational operator. This operator returns true if the operand on the left is less than the operand on the right. For example, “3 LT 4” returns true.

**Using Relational Operators with Arrays**

Relational operators can be applied to arrays, and the resulting array of ones and zeroes can be used as an operand. For example, the expression, `ARR * (ARR LE 100)` is an array equal to ARR except that all points greater than 100 have been

reduced to zero. The expression `(ARR LE 100)` is an array that contains a 1 where the corresponding element of `ARR` is less than or equal to 100, and zero otherwise. For example, to print the number of positive elements in the array `ARR`:

```
PRINT, TOTAL(ARR GT 0)
```

## Conditional Expression

The conditional expression—written with the ternary operator `?:`—has the lowest precedence of all the operators and is used wherever any other expression is used. It provides an alternate way to write simple constructions of the `IF:THEN:ELSE` combination. In the following example, `z` holds the greater value, `a` or `b`. Note that if `a=b`, `z` holds `b`.

```
IF (a GT b) THEN z = a ELSE z = b
```

Using a conditional expression, this statement can be simplified. Set `z` to the greater of `a` and `b`, with `z=b` if `a=b`.

```
z = (a GT b) ? a : b
```

The general form of this expression follows:

```
expr1 ? expr2 : expr3
```

The expression *expr1* is evaluated first. If *expr1* is true, then the expression *expr2* is evaluated and set as the value of the conditional expression. If *expr1* is false, *expr3* is evaluated and set as the value of the conditional expression. Either *expr2* or *expr3* is evaluated, based on the result of *expr1*.

---

### Note

Since `?:` has very low precedence—just above assignment—parentheses are not necessary around the first expression *expr1*. Parentheses are advisable anyway to distinguish the condition part of the expression.

---

For more information about the behavior of the `?:` operator, see [“Definition of True and False”](#) on page 220.

# Type and Structure of Expressions

Every entity in IDL has an associated type and structure. The twelve atomic data types in decreasing order of precedence are as follows:

- Double-precision complex floating-point
- Complex floating-point
- Double-precision floating-point
- Floating-point
- Signed and unsigned 64-bit integer
- Signed and unsigned longword (32-bit) integer
- Signed and unsigned (16-bit) integer
- Byte
- String

The structure of an expression can be either a scalar or an array. The type and structure of an expression depends on the type and structure of its operands. Unlike many other languages, the type and structure of most expressions in IDL cannot be determined until the expression is evaluated. Because of this, care must be taken when writing programs. For example, a variable can be a scalar byte variable at one point in a program while at a later point it can be set to a complex array.

## Expression Type

IDL attempts to evaluate expressions containing operands of different types in the most accurate manner possible. The result of an operation becomes the same type as the operand with the greatest precedence or potential precision. For example, when adding a byte variable to a floating-point variable, the byte variable is first converted to floating-point, then added to the floating-point variable, yielding a floating-point result. When adding a double-precision variable to a complex variable, the result is double precision complex because the double precision complex type has a higher position in the hierarchy of data types.

**Note**


---

Signed and unsigned integers of a given width have the same precedence. In an expression involving a combination of such types, the result is given the type of the *leftmost* operand.

---

When writing expressions with mixed types, care must be taken to obtain the desired results. For example, assume the variable *A* is an integer variable with a value of 5. The following expressions yield the indicated results:

```
;Integer division is performed. The remainder is discarded.
A / 2 = 2
```

```
;The value of A is first converted to floating.
A / 2. = 2.5
```

```
;Integer division is done first because of operator precedence.
;Result is floating point.
A / 2 + 1. = 3.
```

```
;Division is done in floating, then the 1 is converted to floating
;and added.
A / 2. +1 = 3.5
```

```
;Signed and unsigned integer operands have the same precedence, so
;the left-most operand determines the type of the result as signed
;integer.
A + 5U = 10
```

```
;As above, the left-most operand determines the result type
;between types with the same precedence
5U + 1 = 10U
```

**Note**


---

When other types are converted to complex type, the real part of the result is obtained from the original value and the imaginary part is set to zero.

---

When a string type appears as an operand with a numeric data type, the string is converted to the type of the numeric term. For example: '123' + 123.0 is 246.0, while '123.333' + 33 gives the result 156 because 123.333 is first converted to integer type. In the same manner, 'ABC' + 123 also causes a conversion error.



## Expression Structure

IDL expressions can contain operands with different structures, just as they can contain operands with different types. Structure conversion is independent of type conversion. An expression will yield an array result if any of its operands is an array, as shown in the following table:

Operands	Result
Scalar : Scalar	Scalar
Array : Array	Array
Scalar : Array	Array
Array : Scalar	Array

*Table 6-6: Structure of Expressions*

Functions exist to create arrays of the data types IDL supports: `BYTARR`, `INTARR`, `UINTARR`, `LONARR`, `ULONARR`, `LON64ARR`, `ULON64ARR`, `FLTARR`, `DCOMPLEXARR`, `DBLARR`, `COMPLEXARR`, `OBJARR`, `PTRARR`, and `STRARR`. The dimensions of the desired array are the parameters to these functions. The result of `FLTARR(5)` is a floating-point array with one dimension, a vector, with five elements initialized to zero. `FLTARR(50,100)` is a two-dimensional array, a matrix, with 50 columns and 100 rows.

The size of an array-valued expression is equal to the smaller of its array operands. For example, adding a 50-point array to a 100-point array gives a 50-point array; the last 50 points of the larger array are ignored. Array operations are performed point-by-point, without regard to individual dimensions. An operation involving a scalar and an array always yields an array of identical dimensions. When two arrays of equal size (number of elements) but different structure are operands, the result is of the same structure as the first operand. For example:

```
;Yields fltarr(4).
FLTARR(4) + FLTARR(1, 4)
```

In the above example, a row vector is added to a column vector and a row vector is obtained because the operands are the same size. This causes the result to take the structure of the first operand. Here are some examples of expressions involving arrays:

```
;An array in which each element is equal to the same element in ARR
;plus one. The result has the same dimensions as ARR. If ARR is
```

```
;byte or integer, the result is of integer type; otherwise, the  
;result is the same type as ARR.
```

```
ARR + 1
```

```
;An array obtained by summing two arrays.
```

```
ARR1 + ARR2
```

```
;An array in which each element is set to twice the smaller of  
;either the corresponding element of ARR or 100.
```

```
(ARR < 100) * 2
```

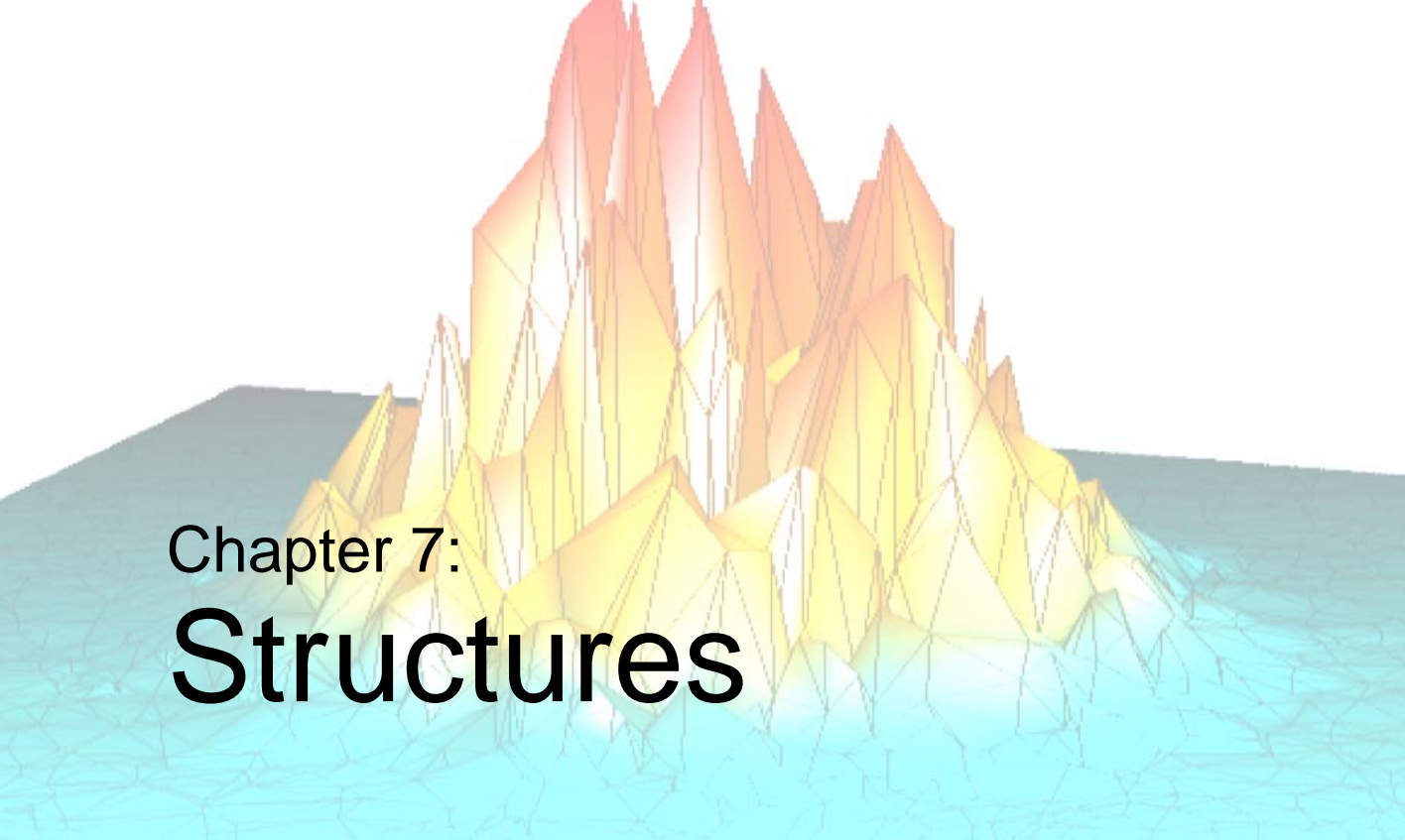
```
;An array in which each element is equal to the exponential of the  
;same element of ARR divided by 10.
```

```
EXP(ARR/10.)
```

```
;An inefficient way of coding ARR * (3./MAX(ARR))
```

```
ARR * 3./MAX(ARR)
```

In the last example, each point in `ARR` is multiplied by three, then divided by the largest element of `ARR`. The `MAX` function returns the largest element of its array argument. This way of writing the statement requires that each element of `ARR` be operated on twice. If `( 3 ./MAX(ARR) )` is evaluated with one division and the result then multiplied by each point in `ARR`, the process requires approximately half the time.



# Chapter 7: Structures

The following topics are covered in this chapter:

---

Overview .....	132	Arrays of Structures .....	143
Creating and Defining Structures .....	133	Structure Input/Output .....	145
Structure References .....	136	Advanced Structure Usage .....	147
Using HELP with Structures .....	139	Automatic Structure Definition .....	149
Parameter Passing with Structures .....	140	Relaxed Structure Assignment .....	151

# Overview

IDL supports structures and arrays of structures. A structure is a collection of scalars, arrays, or other structures contained in a variable. Structures are useful for representing data in a natural form, transferring data to and from other programs, and containing a group of related items of various types. There are two types of structures and they have similar features.

## Named Structures

Each distinct type of named structure is defined by a unique structure name. The first time a structure name is used, IDL creates and saves a definition of the structure which cannot be changed. Each structure definition consists of the structure's name and a definition of each field that is a member of the structure. Each instance of a named structure shares the same definition. Named structures are used when their definitions will not be changed.

## Anonymous Structures

If a structure definition contains no name, an anonymous structure is created. A unique structure definition is created for each anonymous structure. Use anonymous structures when the structure, type, and/or dimensions of its components change during program execution.

Each field definition consists of a tag name and a tag definition that contains the type and structure of the data contained in the field. A field is referred to by its tag name. The tag definition is simply an expression or variable. The type, structure, and value of the tag definition serve to define the field's type, structure, and value. As with structure definitions, a field definition is fixed and cannot be changed. The contents of a field can be any type of data representable by IDL. Fields can contain scalars, arrays of the seven basic data types, and even other structures or arrays of structures.

# Creating and Defining Structures

A named structure is created by executing a structure-definition expression, which is an expression of the following form:

```
{ Structure_Name, Tag_Name1 : Tag_Definition1, ..., Tag_Namen : Tag_Definitionn }
```

Anonymous structures are created in the same way, but with the structure's name omitted.

```
{ Tag_Name1 : Tag_Definition1, ..., Tag_Namen : Tag_Definitionn }
```

Anonymous structures can also be created and combined using the `CREATE_STRUCT` function.

Tag names must be unique within a given structure, although the same tag name can be used in more than one structure. Structure names and tag names follow the rules of IDL identifiers: they must begin with a letter; following characters can be letters, digits, or the underscore or dollar sign characters; and case is ignored.

As mentioned above, each tag definition is a constant, variable, or expression whose structure defines the structure and initial value of the field. The result of the structure definition expression is an instance of the structure, with each field set equal to its tag definition.

A named structure that has already been defined can be referred to by simply enclosing the structure's name in braces, as shown below:

```
{ Structure_Name }
```

The result of this expression is a structure of the designated name.

## Note

---

When a new instance of a structure is created from an existing named structure, all of the fields in the newly-created structure are *zeroed*. This means that fields containing numeric values will contain zeros, fields containing string values will contain null strings, and fields containing pointers or objects will contain null pointers or null objects. In other words, no matter what data the original structure contained, the new structure will contain only a template for that type of data.

---

Also, when making a named structure that has already been defined, the tag names need not be present:

```
{ Structure_Name, expression1, ..., expressionn }
```

All of the expressions must agree in structure with the original tag definition.

*Once defined, a given named structure type cannot be changed.* If a structure definition with tag names is executed and the structure already exists, each tag name and the structure of each tag field must agree with the original definition. Anonymous structures do not have this restriction because each instance has its own definition.

## Structure Inheritance

Structures can inherit tag names and definitions from other structures. To cause one structure to inherit tags from another, use the `INHERITS` specifier. For example, if we define a structure `one` as follows:

```
A = { one, data1a:0, data1b:0L }
```

we can define a second structure `two` that includes the tags from the `one` structure with the following definition statement:

```
B = { two, INHERITS one, data2:0.0 }
```

This is the same as defining the structure `two` with the statement:

```
B = { two, data1a:0, data1b:0L, data2:0.0 }
```

Note that the fields of the `one` structure are included in the `two` structure in the position that the `INHERITS` specifier appears in the structure definition.

Remember that tag names must be unique. If you use structure inheritance, be sure that the tag names in the inherited structure do not conflict with the tag names in the inheriting structure.

Structures that are inherited must be defined before the inheriting structure can be defined. If a structure inherits tags from another structure that is not yet defined, IDL will search for a routine to define the inherited structure as outlined in [“Automatic Structure Definition”](#) on page 149. If the inherited structure cannot be defined, definition of the new structure fails.

While structure inheritance can be used with any structure, it is most useful when dealing with *object class structures*. When the `INHERITS` specifier is used in a class structure definition, it has the added effect of defining the inheriting object as a *subclass* of the inherited class. For a discussion of object-oriented IDL programming, see [Chapter 12, “Object Basics”](#).

## Example of Creating a Structure

Assume that a star catalog is to be processed. Each entry for a star contains the following information: star name, right ascension, declination, and an intensity

measured each month over the last 12 months. A structure for this information is defined with the following IDL statement:

```
A = {star, name:'', ra:0.0, dec:0.0, inten:FLTARR(12)}
```

This structure definition is the basis for all examples in this chapter. The statement above defines a structure type named `star`, which contains four fields. The tag names are `name`, `ra`, `dec`, and `inten`. The first field, with the tag `name`, contains a scalar string as given by its tag definition. The following two fields each contain floating-point scalars. The fourth field, `inten`, contains a 12-element, floating-point array. Note that the type of the constants, `0.0`, is floating point. If the constants had been written as `0`, the fields `ra` and `dec` would contain short integers.

The same structure is created as an anonymous structure by the statement:

```
A = {name:'', ra:0.0, dec:0.0, inten:FLTARR(12)}
```

or by using the `CREATE_STRUCT` function:

```
A = CREATE_STRUCT('name', '', 'ra', 0.0, 'dec', 0.0, $  
    'inten', FLTARR(12))
```

## Structure References

The basic syntax of a reference to a field within a structure is as follows:

*Variable\_Name.Tag\_Name*

*Variable\_Name* must be a variable that contains a structure. *Tag\_Name* is the name of the field and must exist in the structure. If the field referred to by the tag name is itself a structure, the *Tag\_Name* can optionally be followed by one or more additional tag names, as shown by the following example:

```
var .tag1 .tag2
```

This nesting of structure references can be continued up to 10 levels. Each tag name, except possibly the last, must refer to a field that contains a structure.

### Subscripted Structure References

A subscript specification can be appended to the variable or tag names if the variable is an array of structures or if the field referred to by the tag contains an array. Scalar fields within a structure can also be subscripted, provided the subscript is zero.

*Variable\_Name.Tag\_Name[Subscripts]*

*Variable\_Name[Subscripts].Tag\_Name...*

*Variable\_Name[Subscripts].Tag\_Name[Subscripts]*

Each subscript is applied to the variable or tag name it immediately follows. The syntax and meaning of the subscript specification is similar to simple array subscripting in that it can contain a simple subscript, array of subscripts, or a subscript range. If a variable or field containing an array is referenced without a subscript specification, all elements of the item are affected. Similarly, when a variable that contains an array of structures is referenced without a subscript but with a tag name, the designated field in all array elements is affected. The complete syntax of references to structures follows. (Optional items are enclosed in curly brackets, { }.)

*Structure\_reference:= Variable\_Name{[Subscripts]}.Tags*

*Tags:= {Tags.}Tag*

*Tag:= Tag\_Name{[Subscripts]}*

For example, all of the following are valid structure references:



```

A.B
A.B[N, M]
A[12].B
A[3:5].B[* , N]
A[12].B.C[X, *]

```

The semantics of storing into a structure field using subscript ranges is slightly different than that of simple arrays. This is because the structure of arrays in fields are fixed. See [“Storing Into Array Fields”](#) on page 140.

## Examples of Structure References

The name of the star contained in A is referenced as A.NAME. The entire intensity array is referred to as A.INTEN, while the n-th element of A.INTEN is A.INTEN[N]. The following are valid IDL statements using the STAR structure:

```

;Store a structure of type STAR into variable A. Define the values
;of all fields.
A = {star, name:'SIRIUS', ra:30., dec:40., inten:INDGEN(12)}

;Set name field. Other fields remain unchanged.
A.name = 'BETELGEUSE'

;Print name, right ascension, and declination.
PRINT, A.name, A.ra, A.dec

;Set Q to the value of the sixth element of A.inten. Q will be a
;floating-point scalar.
Q = A.inten[5]

;Set ra field to 23.21.
A.ra = 23.21

;Zero all 12 elements of intensity field. Because the type and size
;of A.inten are fixed by the structure definition, the semantics of
;assignment statements is somewhat different than with normal
;variables.
A.inten = 0

;Store fourth through seventh elements of inten field in
;variable B.
B = A.inten[3:6]

;The integer 12 is converted to string and stored in the name field
;because the field is defined as a string.
A.name = 12

```

```
;Copy A to B. The entire structure is copied and B contains a STAR  
;structure.  
B = A
```

## Using HELP with Structures

Use the `HELP, /STRUCTURE` command to determine the type, structure, and tag name of each field in a structure. In the example above, a structure was stored into variable `A`. The statement,

```
HELP, /STRUCTURE, A
```

prints the following information:

```
** Structure STAR, 4 tags, length=40:  
NAME          STRING      'SIRIUS'  
RA            FLOAT        30.0000  
DEC           FLOAT        40.0000  
INTEN        INT          Array(12)
```

Using `HELP` with anonymous structures prints the structure's name as a unique number enclosed in angle brackets. Calling `HELP` with the `STRUCTURE` keyword and no parameters prints a list of all defined, named structure types and their tag names.

## Parameter Passing with Structures

An entire structure is passed by reference by simply using the name of the variable containing the structure as a parameter. Changes to the parameter within the procedure are passed back to the calling procedure. Fields within a structure are passed by value. For example, the following statement prints the value of the structure field `A.name`:

```
PRINT, A.name
```

Any reference to a structure with a subscript or tag name is evaluated into an expression, hence `A.name` is an expression and is passed by value. This works as expected unless the called procedure returns information in the parameter. For example, the call

```
READ, A.name
```

does not read into `A.name` but interprets its parameter as a prompt string. The proper code to read into the field is as follows.

```
;Copy type and attributes to variable.  
B = A.name  
  
;Read into a simple variable.  
READ, B  
  
;Store result into field.  
A.name = B
```

### Storing Into Array Fields

As mentioned previously, the semantics of storing into structure array fields is slightly different than storing into simple arrays. The main difference is that with structures, a subscript range must be used when storing an array into part of an array field. With normal arrays, when storing an array inside part of another array, use the subscript of the lower-left corner, not a range specification. Other differences occur because the size and type of a field are fixed by the original structure definition, and the normal IDL semantics of dynamic binding are not applicable. The rules for storing into array fields are as follows:

**VAR.ARRAY\_TAG** = *Scalar\_Expression*

All elements of VAR.tag are set to *Scalar\_Expression*. For example:

```
;Set all 12 elements of A.inten to 100.
A.inten = 100
```

**VAR.TAG** = *Array\_Expression*

Each element of *Array\_Expression* is copied into the array VAR.tag. If *Array\_Expression* contains more elements than the destination array does, an error results. If it contains fewer elements than VAR.TAG, the unmatched elements remain unchanged. For example:

```
;Set A.inten to the 12 numbers 0, 1, 2, ..., 11.
A.inten = FINDGEN(12)

;Set A.inten[0] to 1 and A.inten[1] to 2. The other elements
;remain unchanged.
A.inten = [1, 2]
```

**VAR.TAG[Subscript]** = *Scalar\_Expression*

The value of the scalar expression is simply copied into the designated element of the destination. If *Subscript* is an array of subscripts, the scalar expression is copied into the designated elements. For example:

```
;Set the sixth element of A.inten to 100.
A.inten[5] = 100

;Set elements 2, 4, and 6 to 100.
A.inten[[2, 4, 6]] = 100
```

**VAR.TAG[Subscript]** = *Array\_Expression*

Unless VAR.tag is an array of structures, the subscript must be an array. Each element of *Array\_Expression* is copied into the element given by the corresponding element subscript. For example:

```
;Set elements 2, 4, and 6 to the values 5, 7, and 9 respectively.
A.inten[[2, 4, 6]] = [5, 7, 9]
```

**VAR.TAG[Subscript\_Range]** = *Scalar\_Expression*

The value of the scalar expression is stored into each element specified by the subscript range. For example:

```
;Sets elements 8, 9, 10, and 11 to the value 5.
A.inten[8:*] = 5
```

**VAR.TAG**[*Subscript\_Range*] = *Array\_Expression*

Each element of the array expression is stored into the element designated by the subscript range. The number of elements in the array expression must agree with the size of the subscript range. For example:

```
;Sets elements 3, 4, 5, and 6 to the numbers 0, 1, 2, and 3,  
;respectively.  
A.inten[3:6] = FINDGEN(4)
```

# Arrays of Structures

An array of structures is simply an array in which each element is a structure of the same type. The referencing and subscripting of these arrays (also called structure arrays) follow the same rules as simple arrays.

## Creating an Array of Structures

The easiest way to create an array of structures is to use the REPLICATE function. The first parameter to REPLICATE is a reference to the structure of each element. Using the example in “[Examples of Structure References](#)” on page 137 and assuming the STAR structure has been defined, an array containing 100 elements of the structure is created with the following statement:

```
cat = REPLICATE({star}, 100)
```

Alternatively, since the variable A contains an instance of the structure STAR, then

```
cat = REPLICATE(A, 100)
```

Or, to define the structure and an array of the structure in one step, use the following statement:

```
cat = REPLICATE({star, name:'', ra:0.0, dec:0.0, $
    inten:FLTARR(12)}, 100)
```

The concepts and combinations of subscripts, subscript arrays, subscript ranges, fields, nested structures, etc., are quite general and lead to many possibilities, only a small number of which can be explained here. In general, any structures that are similar to the examples above are allowed.

## Examples of Arrays of Structures

This example uses the above definition in which the variable CAT contains a star catalog of STAR structures.

```
;Set the name field of all 100 elements to "EMPTY."
cat.name = 'EMPTY'

;Set the i-th element of cat to the contents of the star structure.
cat[I] = {star, 'BETELGEUSE', 12.4, 54.2, FLTARR(12)}

;Store 0.0 into cat[0].ra, 1.0 into cat[1].ra, ..., 99.0 into
;cat[99].ra
cat.ra = INDGEN(100)
```

```
;Prints name field of all 100 elements of cat, separated by commas
;(the last field has a trailing comma).
PRINT, cat.name + ', '

;Find index of star with name of SIRIUS.
I = WHERE(cat.name EQ 'SIRIUS')

;Extract intensity field from each entry. Q will be a 12 by 100
;floating-point array.
Q = cat.inten

;Plot intensity of sixth star in array cat.
PLOT, cat[5].inten

;Make a contour plot of the (7,46) floating-point array ;taken from
;months (2:8) and stars (5:50).
CONTOUR, cat[5:50].inten[2:8]

;Sort the array into ascending order by names. Store the result
;back into cat.
cat = cat(SORT(cat.name))

;Determine the monthly total intensity of all stars in array.
;monthly is now a 12-element array.
monthly = cat.inten # REPLICATE(1,100)
```



# Structure Input/Output

Structures are read and written using the formatted and unformatted input/output procedures `READ`, `PRINT`, `READU`, and `WRITEU`. Structures and arrays of structures are transferred in much the same way as simple data types, with each element of the structure transferred in order.

## Formatted Input/Output with Structures

Writing a structure with `PRINT` or `PRINTF` and the default format outputs the contents of each element using the default format for the appropriate data type. The entire structure is enclosed in braces: “{ }”. Each array begins a new line. For example, printing the variable `A`, as defined in the first example in this chapter, results in the following output.

```
{SIRIUS 30.0000 40.0000 0 1 2 3 4 5 6 7 8 9 10 11}
```

When reading a structure with `READ` or `READF` and the default format, white space should separate each element. Reading string elements causes the remainder of the input line to be stored in the string element, regardless of spaces, etc. A format specification can be used with any of these procedures to override the default formats. The length of string elements is determined by the format specification (i.e., to read the next 10 characters into a string field, use an `(A10)` format).

## Unformatted Input/Output with Structures

Reading and writing unformatted data contained in structures is a straightforward process of transferring each element, without interpretation or modification, except in the case of strings. Each IDL data type, except strings, has a fixed length expressed in bytes. This length (which is padded when using `ASSOC`, but *not* padded when using `READU`/`WRITEU`) is also the number of bytes read or written for each element.

All instances of structures contain an even number of bytes. On machines whose native C compilers force short integers to begin on an even byte boundary, IDL begins fields that are not of type byte on an even byte boundary. Thus, a “padding byte” may appear (when using `ASSOC` for I/O) after a byte field to cause the following non-byte-type field to begin on an even byte. A padding byte is never added before a byte or byte array field. For example, the structure:

```
{example, t1:1b, t2:1}
```

occupies four bytes on a machine where short integers must begin on an even byte boundary. When using ASSOC, a padding byte is added after field `t1` to cause the integer field `t2` to begin on an even-byte boundary.

## Strings

Strings are exceptions to the above rules because the length of strings within structures is not fixed. For example, one instance of the `{star}` structure can contain a `name` field with a five-character name, while another instance of the same structure can contain a 20-character name. When reading into a structure field that contains a string, IDL reads the number of bytes given by the length of the string. If the string field contains a 10-character string, 10 characters are read. If the data read contains a null byte, the length of the string field is truncated, and the null and following characters are discarded. When writing fields containing strings with the unformatted procedure `WRITEU`, IDL writes each character of the string and does not append a terminating null byte.

## String Length Issues

When reading or writing structures containing strings with `READU` and `WRITEU`, make each string in a given field the same length to be compatible with C and to be able to read the data back into IDL. You must know how many characters exist to read into a string element. One way around this problem is using the `STRING` function with a format specification that sets the length of all elements to some maximum number. For example, it is easy to set the length of all `name` fields in the `cat` array to 20 characters by using the following statement.

```
cat.name = STRING(cat.name, FORMAT = '(A20)')
```

This statement will truncate names longer than 20 characters and will pad with blanks those names shorter than 20 characters. The structure or structure array then can be output in a format suitable to be read by C or FORTRAN programs. For example, to read into the `cat` array from a file in which each `name` field occupies 26 bytes, use the following statements.

```
;Make a 100-element array of {STAR} structures, storing a
;26-character string in each name field.
cat = REPLICATE({star, STRING(' ', FORMAT = '(A26)'), $
  FLTARR(0., 0.12)}, 100)
```

```
;Read the structure. As mentioned above, 26 bytes will be read for
;each name field. The presence of a null byte in the file will
;truncate the field to the correct number of bytes.
READU, 1, cat
```

# Advanced Structure Usage

Facilities exist to process structures in a general way using tag *numbers* rather than tag names. A tag can be referenced using its index, enclosed in parentheses, as follows:

```
Variable_Name.(Tag_Index)... ..
```

The *Tag\_Index* ranges from zero to the number of fields minus one.

---

## Note

The *Tag\_Index* is an expression, the result of which is taken to be a tag position. In order for the IDL parser to understand that this is the case, you must enclose the *Tag\_Index* in parentheses. This is not an array indexing operation, so the use of square brackets ([]) is not allowed in this context.

---

## Number of Structure Tags

The function `N_TAGS(Structure)` returns the number of fields in a structure. To obtain the size, in bytes, of a structure call `N_TAGS` with the `/LENGTH` keyword.

## Names of Structure Tags

The function `TAG_NAMES(Structure)` returns a string array containing the names of each tag. To return the name of the structure itself, call `TAG_NAMES` with the `/STRUCTURE_NAME` keyword.

## Example

Using tag indices and the above-mentioned functions, we specify a procedure that reads into a structure from the keyboard. The procedure prompts the user with the type, structure, and tag name of each field within the structure.

```

;A procedure to read into a structure, S, from the keyboard with
;prompts.
PRO READ_STRUCTURE, S

;Get the names of the tags.
NAMES = TAG_NAMES(S)
;Loop for each field.
FOR I = 0, N_TAGS(S) - 1 DO BEGIN
    ;Define variable A of same type and structure as the i-th field.
    A = S.(I)

```

```
        ;Use HELP to print the attributes of the field. Prompt user with
        ;tag name of this field, and then read into variable A. S.(I) =
        ;A. Store back into structure from A.
        HELP, S.(I)

        READ, 'Enter Value For Field ', NAMES[I], ': ', A
        S.(I) = A
    ENDFOR
END
```

---

**Note**

In the above procedure, the READ procedure reads into the variable A rather than S.(I) because S.(I) is an expression, not a simple variable reference. Expressions are passed by value; variables are passed by reference. The READ procedure prompts the user with parameters passed by value and reads into parameters passed by reference.

---

## Automatic Structure Definition

In versions of IDL prior to version 5, references to an undefined named structure would cause IDL to halt with an error. This behavior was changed in IDL version 5 to allow the automatic definition of named structures.

When IDL encounters a reference to an undefined named structure, it will automatically search the directories specified in `!PATH` for a procedure named `Name__DEFINE`, where `Name` is the actual name of the structure. If this procedure is found, IDL will call it, giving it the opportunity to define the structure. If the procedure does in fact define the named structure, IDL will proceed with the desired operation.

### Note

---

There are *two* underscores in the name of the structure definition procedure.

---

For example, suppose that a structure named `mystruct` has not been defined, and that no procedure named `mystruct__define.pro` exists in the directories specified by `!PATH`. A call to the `HELP` procedure produces the following output:

```
HELP, { mystruct }, /STRUCTURE
```

IDL prints:

```
% Attempt to call undefined procedure/function: 'MYSTRUCT__DEFINE'.
% Structure type not defined: MYSTRUCT.
% Execution halted at: $MAIN$
```

Suppose now that we define a procedure named `mystruct__define.pro` as follows, and place it in one of the directories specified by `!PATH`:

```
PRO mystruct__define
  tmp = { mystruct, a:1.0, b:'string' }
END
```

With this structure definition routine available, the call to `HELP` produces the following output:

```
HELP, { mystruct }, /STRUCTURE
```

IDL prints:

```
% Compiled module: MYSTRUCT__DEFINE.
** Structure MYSTRUCT, 2 tags, length=12:
  A          FLOAT          0.00000
  B          STRING         ''
```

Remember that the fields of a structure created by copying a named structure definition are filled with zeroes or null strings. Any structure created in this way—either via automatic structure definition or by explicitly creating a new structure from an existing structure—must be initialized to contain values after creation.

## Relaxed Structure Assignment

The IDL “=” operator is unable to assign a structure value to a structure with a different definition. For example, suppose we have an existing structure definition SRC, as follows:

```
source = { SRC, A:FINDGEN(4), B:12 }
```

and we wish to create a second instance of the same structure, but with slightly different data and a different field:

```
dest = { SRC, A:INDGEN(2), C:20 }
```

Attempting to execute these two statements at the IDL command prompt gives the following results:

```
% Conflicting data structures: <INT      Array[2]>,SRC.
% Execution halted at: $MAIN$
```

Versions of IDL beginning with IDL 5.1 include a mechanism to solve this problem. The STRUCT\_ASSIGN procedure performs “relaxed structure assignment,” which is a field-by-field copy of a structure to another structure. Fields are copied according to the following rules:

1. Any fields found in the destination structure that are not found in the source structure are “zeroed” (set to zero, the empty string, or a null pointer or object reference depending on the type of field).
2. Any fields in the source structure that are not found in the destination structure are quietly ignored.
3. Any fields that are found in both the source and destination structures are copied one at a time. If necessary, type conversion is done to make their types agree. If a field in the source structure has fewer data elements than the corresponding field in the destination structure, then the “extra” elements in the field in the destination structure are zeroed. If a field in the source structure has more elements than the corresponding field in the destination structure, the extra elements are quietly ignored.

Using STRUCT\_ASSIGN, we can make the assignment that failed using the = operator:

```
source = { src, a:FINDGEN(4), b:12 }
dest = { dest, a:INDGEN(2), c:20 }
STRUCT_ASSIGN, source, dest, /VERBOSE
```

IDL prints:

```
% STRUCT_ASSIGN: SRC tag A is longer than destination.
                    The end will be clipped.
% STRUCT_ASSIGN: Destination lacks SRC tag B. Not copied.
```

If we check the variable `dest`, we see that it has the definition of the `dest` structure and the data from the `source` structure:

```
HELP, dest, /STRUCTURE
```

IDL prints:

```
** Structure DEST, 2 tags, length=6:
   A           INT           Array[2]
   C           INT           0
```

## Using Relaxed Structure Assignment

Why would you want to use Relaxed Structure Assignment? One case where this type of structure definition is very useful is in restoring object structures into an environment where the structure definition may have changed since the restored objects were saved.

Suppose you have created an application that saves data in structures. Your application may use the IDL `SAVE` routine to save the data structures to disk files. If you later change your application such that the definition of the data structures changes, you would not be able to restore your saved data into your application's framework without relaxed structure assignment. The `RELAXED_STRUCTURE_ASSIGNMENT` keyword to the `RESTORE` procedure allows you to make relaxed assignments in such cases.

To see how this works, try the following exercise:

1. Start IDL, create a named structure, and use the `SAVE` procedure to save it to a file:

```
mystruct = { STR, A:10, B:20L, C:'a string' }
SAVE, mystruct, FILE='test.dat'
```

2. Exit and restart IDL.
3. Create a new structure definition with the same name you used previously:

```
newstruct = { STR, A:20L, B:10.0, C:'a string', D:ptr_new() }
```

4. Attempt to restore the variable `mystruct` from the `test.dat` file:

```
RESTORE, 'test.dat'
```

IDL prints:



```
% Wrong number of tags defined for structure: STR.
% RESTORE: Structure not restored due to conflict with
    existing definition: STR.
```

5. Now use relaxed structure definition when restoring:

```
RESTORE, 'test.dat', /RELAXED_STRUCTURE_ASSIGNMENT
```

6. Check the contents of mystruct:

```
HELP, mystruct, /STRUCTURE
```

IDL prints:

```
** Structure STR, 4 tags, length=20:
  A          LONG          10
  B          FLOAT         20.0000
  C          STRING       'a string'
  D          POINTER      <NullPointer>
```

The structure in the variable `mystruct` now uses the definition from the new version of the `STR` structure, but contains the data from the old (restored) structure. In cases where the data type of a field has changed, the data type of the old data element has been converted to the new data type. Fields in the new structure definition that do not correspond to fields in the old definition contain “zero” values (zeroes for numeric fields, empty strings for string fields, null pointer or references for pointer or reference fields).





# Chapter 8: Array Subscripts

The following topics are covered in this chapter:

---

Overview .....	156	Structure of Subarrays .....	163
Array Subscript Syntax: [ ] vs. ( ) .....	157	Array Subscripts .....	165
Subscript Examples .....	158	Combining Array Subscripts with Others .	167
Subscript Ranges .....	161	Storing Elements with Array Subscripts ...	169

## Overview

Subscripts provide a means of selecting one or more elements of an array for retrieval or modification.

The values of the selected array elements are extracted when a subscripted variable reference appears in an expression. New values are stored in selected array elements, without disturbing the remaining elements, when a subscript reference appears on the left side of an assignment statement. “[The Assignment Statement](#)” on page 198 discusses the use of the different types of assignment statements when storing into arrays.

The subscripts of an array element denote the address of the element within the array. In the simple case of a one-dimensional array, an  $n$ -element vector, elements are numbered starting at 0 with the first element, 1 for the second element, and running to  $n - 1$ , the subscript of the last element.

Arrays with multiple dimensions are addressed by specifying a subscript expression for each dimension. A two-dimensional array, a matrix with  $n$  columns and  $m$  rows, is addressed with a subscript of the form  $[i, j]$ , where  $0 \leq i < n$  and  $0 \leq j < m$ . The first subscript,  $i$ , is the column index; the second subscript,  $j$ , is the row index. The syntax of a subscript reference is:

*Variable\_Name* [*Subscript\_List*]

or

*(Array\_Expression)*[*Subscript\_List*]

The *Subscript\_List* is simply a list of expressions, constants, or subscript ranges containing the values of one or more subscripts. Subscript expressions are separated by commas if there is more than one subscript. In addition, multiple elements are selected with subscript expressions that contain either a contiguous range of subscripts or an array of subscripts.

## Array Subscript Syntax: [ ] vs. ( )

Versions of IDL prior to version 5.0 used parentheses to indicate array subscripts. Function calls use parentheses in a visually identical way to specify argument lists. As a result, the IDL compiler is not able to distinguish between arrays and functions by looking at the statement syntax. For example, the IDL statement

```
value = fish(5)
```

could either set the variable `value` equal to the sixth element of an array named `fish`, or set `value` equal to the result of passing the argument `5` to a function called `fish`.

To determine if it is compiling an array subscript or a function call, IDL checks its internal table of known functions. If it finds a function name that matches the unknown element in the command (`fish`, in the above example), it calls that function with the argument specified. If IDL does not find a function with the correct name in its table of known functions, it assumes that the unknown element is an array, and attempts to return the value of the designated element of that array. This rule generally gives the desired result, but it can be fooled into the wrong choice under certain circumstances, much to the surprise of the unwary programmer.

For this reason, versions of IDL beginning with version 5.0 use square brackets rather than parentheses for array subscripting. An array subscripted in this way is unambiguously interpreted as an array under all circumstances. In IDL 5.0 and later:

```
value = fish[5]
```

sets `value` to the sixth element of an array named `fish`.

Due to the large amount of existing IDL code written in the older syntax, as well as the ingrained habits of thousands of IDL users, IDL continues to allow the old syntax to be used, subject to the ambiguity mentioned above. That is, while

```
value = fish[5]
```

is unambiguous,

```
value = fish(5)
```

is still subject to the same ambiguity—and rules—that applied in IDL versions prior to version 5.0

Since the older syntax has been used widely, you should not be surprised to see it from time to time. However, square brackets are the preferred form, and should be used for new code.

## Subscript Examples

Subscripts can be used either to retrieve the value of one or more array elements or to designate array elements to receive new values. The expression `ARR[12]` denotes the value of the 13th element of `ARR` (because subscripts start at 0), while the statement `ARR[12] = 5` stores the number 5 in the 13th element of `ARR` without changing the other elements.

Elements of multidimensional arrays are specified by using one subscript for each dimension. In arrays and images, the first subscript denotes the column and the second subscript is the row. For matrices, the first subscript denotes the row and the second subscript is the column.

If `A` is a 2-element by 3-element array, the elements are stored in memory as follows:

		Stored in Memory
$A_{0,0}$	$A_{1,0}$	Lowest memory address
$A_{0,1}$	$A_{1,1}$	· · ·
$A_{0,2}$	$A_{1,2}$	Highest memory address

*Table 8-1: Storage of IDL Array Elements in Memory*

The elements are ordered in memory as:  $A_{0,0}$ ,  $A_{1,0}$ ,  $A_{0,1}$ ,  $A_{1,1}$ ,  $A_{0,2}$ ,  $A_{1,2}$ , etc. Thus, IDL arrays are *row major* (i.e., stored by rows). This ordering is like FORTRAN. It is the opposite of the way C and Pascal handle arrays. IDL uses row major storage because it is oriented toward image processing while the other languages stress matrix computation. For a more extensive discussion of row versus column majority and how it relates to IDL mathematics routines, see “[Arrays and Matrices](#)” in Chapter 16 of the *Using IDL* manual.

Images are usually displayed with row zero at the bottom of the screen, matching the display’s coordinate system, although this order can be reversed by setting the system variable `!ORDER` to a nonzero value. Arrays are printed with the first row on top.

Elements of multidimensional arrays also can be specified using only one subscript, in which case the array is treated as a vector with the same number of points. In the above example, `A[2]` is the same element as `A [0, 1]`, and `A[5]` is the same element as `A[1, 2]`.

If an attempt is made to reference a nonexistent element of an array using a scalar subscript (a subscript that is negative or larger than the size of the dimension minus 1), an error occurs and program execution stops.

Subscripts can be any type of vector or scalar expression. If a subscript expression is not integer, a longword integer copy is made and used to evaluate the subscript.

## “Extra” Dimensions

When creating arrays, IDL eliminates all size 1, or “degenerate”, trailing dimensions. Thus, the statements

```
A = INTARR(10, 1)
HELP, A
```

print the following:

```
A                INT                = Array(10)
```

This removal of superfluous dimensions is usually convenient, but it can cause problems when attempting to write fully general procedures and functions. Therefore, IDL allows you to specify “extra” dimensions for an array as long as the extra dimensions are all zero. For example, consider a vector defined as follows:

```
ARR = INDGEN(10)
```

The following are all valid references to the sixth element of ARR:

```
X = ARR[5]
X = ARR[5, 0]
X = ARR[5, 0, 0, *, 0]
```

Thus, the automatic removal of degenerate trailing dimensions does not cause problems for routines that attempt to access the resulting array.

## Subscripting Scalars

Scalar quantities in IDL can be thought of as arrays with dimensions of (1,0). They can be subscripted with a zero reflecting the first and only position. Therefore,

```
;Assign the value of 5 to A.
A = 5

;Print the value of the first element of A.
PRINT, A[0]
```

IDL prints:

```
5
```

If we redefine the first element of A:

```
;Redefine the first element of A.  
A[0] = 6  
  
PRINT, A
```

IDL prints:

```
6
```

---

**Note**

You cannot subscript a variable that has not yet been defined. Thus, if the variable B has not been previously defined, the statement:

```
B[0] = 9
```

will fail with the error “variable is undefined.”

---



# Subscript Ranges

Subscript ranges are used to select a subarray from an array by giving the starting and ending subscripts of the subarray in each dimension. Subscript ranges can be combined with scalar and array subscripts and with other subscript ranges. Any rectangular portion of an array can be selected with subscript ranges. There are four types of subscript ranges:

- A range of subscripts, written  $[e_0:e_1]$ , denoting all elements whose subscripts range from the expression  $e_0$  through  $e_1$  ( $e_0$  must not be greater than  $e_1$ ). For example, if the variable VEC is a 50-element vector, VEC[5:9] is a five-element vector composed of VEC[5] through VEC[9].
- All elements from a given element to the last element of the dimension, written as  $[e:*$ ]. Using the above example, VEC[10:\*) is a 40-element vector made from VEC[10] through VEC[49].
- A simple subscript,  $[n]$ . When used with multidimensional arrays, simple subscripts specify only elements with subscripts equal to the given subscript in that dimension.
- All elements of a dimension, written  $[*]$ . This form is used with multidimensional arrays to select all elements along the dimension. For example, if ARR is a 10-column by 12-row array, ARR[\* , 11] is the last row of ARR, composed of elements [ARR[0,11], ARR[1,11], ..., ARR[9,11]], and is a 10-element row vector. Similarly, ARR[0 , \*) is the first column of ARR, [ARR[0,0], ARR[0,1],..., ARR[0,11]], and its dimensions are 1 column by 12 rows.

Multidimensional subarrays can be specified using any combination of the above forms. For example, ARR[ \* , 0 : 4 ] is made from all columns of rows 0 to 4 of ARR or a 10-column, 5-row matrix. The table below summarizes the possible forms of subscript ranges:

Form	Description
$e$	A simple subscript expression
$e_0:e_1$	Subscript range from $e_0$ to $e_1$
$e:*$	All points from element $e$ to end

Table 8-2: Subscript Ranges

<b>Form</b>	<b>Description</b>
*	All points in the dimension

*Table 8-2: Subscript Ranges*

## Structure of Subarrays

The dimensions of the extracted subarray are determined by the size in each dimension of the subscript range expression. In general, the number of dimensions is equal to the number of subscripts and subscript ranges. The size of the  $n$ -th dimension is equal to one if a simple subscript was used to specify that dimension in the subscript; otherwise, it is equal to the number of elements selected by the corresponding range expression.

Degenerate dimensions (trailing dimensions whose size is equal to one) are removed. This was illustrated in the previous example by the expression `ARR[* , 11]` which resulted in a row vector with a single dimension because the last dimension of the result was one and was removed. On the other hand, the expression `ARR[0 , *]` became a column vector with dimensions of `[1, 12]` showing that the structure of columns is preserved because the dimension with a size of one does not appear at the end.

Using the examples of `VEC`, a 50-element vector, and `A`, a 10-column by 12-row array, some typical subscript range expressions are as follows:

```

;Elements 5 through 10 of VEC, a six-element vector.
VEC[5:10]

;A three-element vector.
VEC[I - 1:I + 1]

;The same vector.
[VEC[I - 1], VEC[I], VEC[I + 1]]

;Elements of VEC from VEC(4) to the end, a 46-element (50 - 4)
;vector.
VEC[4:*]

;The fourth column of A, a 1 column by 12 row vector.
A[3, *]

;The first row of A, a 10-element row vector. Note, the last
;dimension was removed because it was degenerate.
[A[3, 0], A[3, 1], ..., A[3, 11]]
A[* , 0]

;The nine-point neighborhood surrounding A[X,Y], a 3 by 3 array.
A[X - 1:X + 1, Y - 1:Y + 1]

;Three columns of A, a 3 by 12 subarray:
A[3:5,*]

```

See “[The Assignment Statement](#)” on page 198 for a description of the process of assigning values to subarrays.

# Array Subscripts

Arrays can be used as subscripts to other arrays. Each element in the array used as a subscript selects an element in the subscripted array. When used with subscript ranges, more than one element may be selected for each subscript element.

If no subscript ranges are present, the length and structure of the result is the same as that of the subscript expression. The type of the result is the same as that of the subscripted array. If only one subscript is present, all subscripts are interpreted as if the subscripted array has one dimension.

In the simple case of only one subscript, in which the subscript is an array, the process can be written as follows:

$$V(S) = \begin{cases} V_{S_i} & \text{if } 0 \leq S_i < n \\ V_0 & \text{if } S_i < 0 \\ V_{n-1} & \text{if } S_i \geq n \end{cases} \quad \text{for } 0 \leq i < m$$

The vector  $V$  has  $n$  elements, and  $S$  has  $m$  elements. The result  $V(S)$  has the same structure and number of elements as does the subscript vector  $S$ .

If an element of the subscript array is less than or equal to zero, the first element of the subscripted variable is selected. If an element of the subscript is greater than or equal to the last subscript in the subscripted variable ( $N$ , above), the last element is selected.

## Example

As an example, consider the commands:

```
A = [6, 5, 1, 8, 4, 3]
B = [0, 2, 4, 1]
C = A[B]
PRINT, C
```

that produce the following output:

```
6 1 4 5
```

The first element of C is 6 because that is the number in the 0 position of A. The second is 1 because the value in B of 2 indicates the third position in A, and so on.

As another example, assume the variable A is a 10 by 10 array. The expression `A[INDGEN(10) * 11]` yields a 10-element vector equal to the diagonal elements of A. The subscripts of the diagonal elements, `A[0,0]`, `A[1,1]`, ..., `A[9, 9]` are equal to 0, 11, 22, 99, when singularly subscripted. The elements of the vector `INDGEN(10)*11` also are equal to 0, 11, 22, ..., 99. Applying the vector as a subscript selects the diagonal elements.

The WHERE function, which returns a vector of subscripts, can be used to select elements of an array using expressions similar to `A[WHERE(A GT 0)]` which results in a vector composed only of the elements of A that are greater than 0.

# Combining Array Subscripts with Others

Array subscripts can be combined with subscript ranges, simple scalar subscripts and other array subscripts.

When IDL encounters a multidimensional subscript that contains one or more subscript arrays, it builds an array of subscripts by processing each subscript from left to right. The resulting array of subscripts is then applied to the variable that is to be subscripted. As with other subscript operations, trailing degenerate dimensions (those with a size of 1) are eliminated.

## Subscript Ranges

When combining an array subscript with a subscript range, the result is an array of subscripts constructed by combining each element of the subscript array with each member of the subscript range. Combining an  $n$ -element array with an  $m$ -element subscript range yields an  $nm$ -element subscript. Each dimension of the result is equal to the number of elements in the corresponding subscript array or range.

For example, the expression `A[[1, 3, 5], 7:9]` is a nine-element,  $3 \times 3$  array composed of the following elements:

$$\begin{bmatrix} A_{1,7} & A_{3,7} & A_{5,7} \\ A_{1,8} & A_{3,8} & A_{5,8} \\ A_{1,9} & A_{3,9} & A_{5,9} \end{bmatrix}$$

Each element of the three-element subscript array (1, 3, 5) is combined with each element of the three-element range (7, 8, 9).

Another example shows the common process of zeroing the edge elements of a two-dimensional  $n \times m$  array:

```
;Zero the first and last rows.
A[* , [0, M-1]] = 0

;Zero the first and last columns.
A[[0, N - 1], *] = 0
```

## Other Subscript Arrays

When combining two subscript arrays, each element of the first array is combined with the corresponding element of the other subscript array. The two subscript arrays

must have the same number of elements. The resulting subscript array has the same number of elements as its constituents. For example, the expression `A[[1, 3], [5, 9]]` yields the elements `A[1,5]` and `A[3,9]`.

## Scalars

Combining an  $n$ -element subscript range or  $n$ -element subscript array with a scalar yields an  $n$ -element result. The value of the scalar is combined with each element of the range or array. For example, the expression `A[[1, 3, 5], 8]` yields the three-element vector composed of the elements `A[1,8]`, `A[3,8]`, and `A[5,8]`. The second dimension of the result is 1 and is eliminated because it is degenerate. The expression `A[8, [1, 3, 5]]` is the  $1 \times 3$ -column vector `A[8,1]`, `A[8,3]`, and `A[8,5]`, illustrating that leading dimensions are not eliminated.



# Storing Elements with Array Subscripts

One or more values can be stored in selected elements of an array by using an array expression as a subscript for the array on the left side of an assignment statement. Values are taken from the expression on the right side of the assignment statement and stored in the elements whose subscripts are given by the array subscript. The right-hand expression can be either a scalar or array.

The subscript array is converted to longword type before use if necessary. Regardless of structure, this subscript array is interpreted as a vector. For details and examples of storing with vector subscripts, see [“The Assignment Statement”](#) on page 198.

## Examples

The statement:

```
A[[2, 4, 6]] = 0
```

zeroes elements A[2], A[4], and A[6], without changing other elements of A. The statement:

```
A[[2, 4, 6]] = [4, 16, 36]
```

stores 4 in A[2], 16 in A[4], and 36 in A[6].

One way to create a square  $n \times n$  identity matrix is as follows:

```
A = FLTARR(N, N)
A[INDGEN(N) * (N + 1)] = 1.0
```

The expression  $\text{INDGEN}(N) * (N + 1)$  results in a vector containing the subscripts of the diagonal elements  $[0, N+1, 2N+2, \dots, (N-1)*(N+1)]$ . Yet another way is to use two array subscripts. The statements:

```
A = FLTARR(N, N)
A[INDGEN(N), INDGEN(N)] = 1.0
```

create the array subscripts  $[[0,0], [1,1], \dots, [n-1, n-1]]$ . The statement:

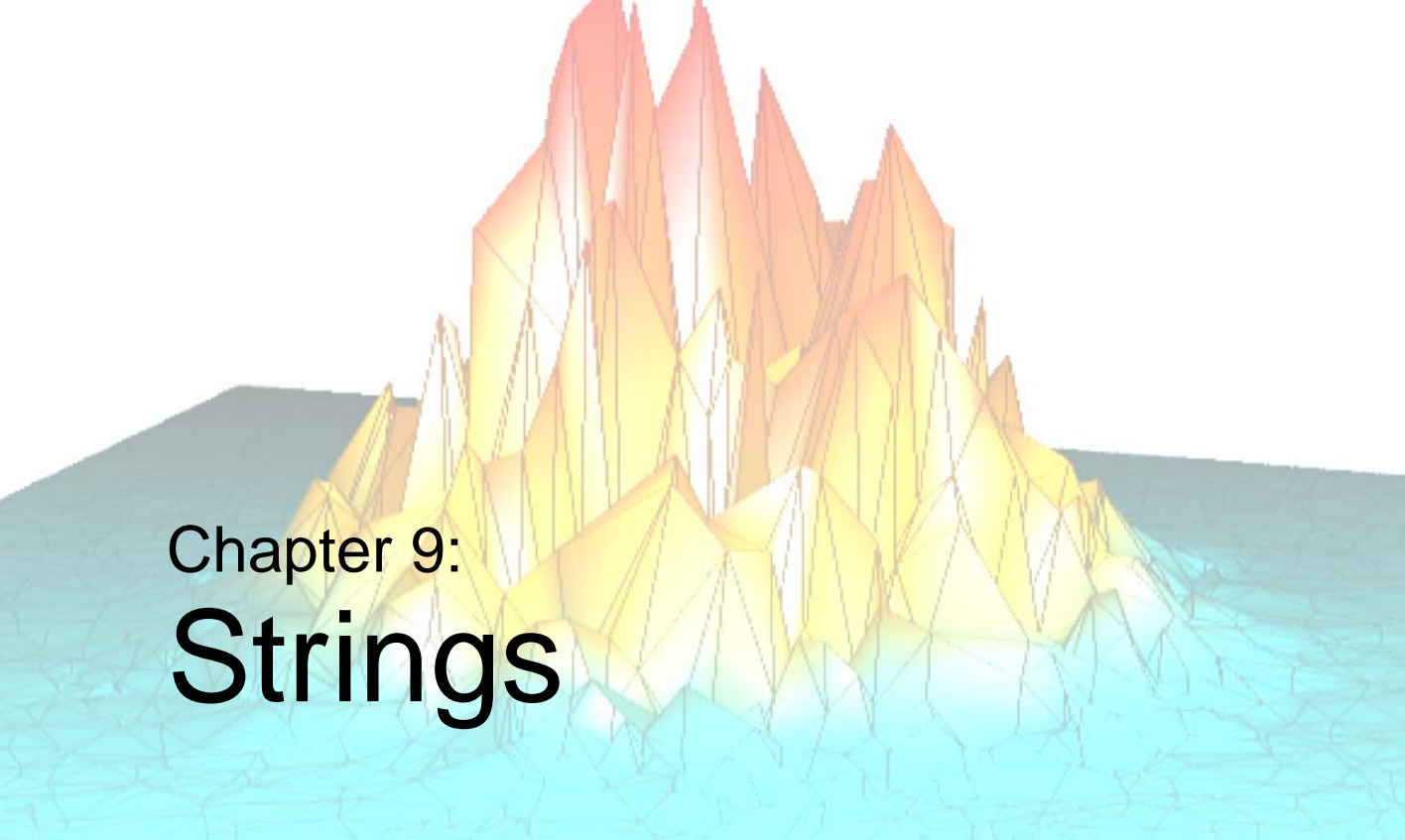
```
A[WHERE(A LT 0)] = -1
```

sets negative elements of A to -1.

The following statements create a 10x10 identity matrix:

```
A = FLTARR(10, 10)
A[INDGEN(10) * 11] = 1
```





# Chapter 9: Strings

The following topics are covered in this chapter:

---

Overview .....	172	Whitespace .....	180
String Operations .....	173	Finding the Length of a String .....	182
Non-string and Non-scalar Arguments ...	174	Substrings .....	183
String Concatenation .....	175	Splitting and Joining Strings .....	186
Using <code>STRING</code> to Format Data .....	176	Comparing Strings .....	187
Byte Arguments and Strings .....	177	Learning About Regular Expressions ....	191
Case Folding .....	179		

# Overview

An IDL string is a sequence of characters from 0 to 32,767 characters in length. Strings have dynamic length (they grow or shrink to fit), and there is no need to declare the maximum length of a string prior to using it. As with any data type, string arrays can be created to hold more than a single string. In this case, the length of each individual string in the array depends only on its own length and is not affected by the lengths of the other string elements.

## A Note About the Examples

In some of the examples in this chapter, it is assumed that a string array named TREES exists. TREES contains the names of seven trees, one name per element, and is created using the statement:

```
trees = ['Beech', 'Birch', 'Mahogany', 'Maple', 'Oak', $
        'Pine', 'Walnut']
```

Executing the statement,

```
PRINT, '>' + trees + '< '
```

results in the following output:

```
>Beech< >Birch< >Mahogany< >Maple< >Oak< >Pine< >Walnut<
```

# String Operations

IDL supports several basic string operations, as described below.

## Concatenation

The [Addition](#) operator, “+”, can be used to concatenate strings together.

## Formatting Data

The [STRING](#) function is used to format data into a string. The [READS](#) procedure can be used to read values from a string into IDL variables.

## Case Folding

The [STRLOWCASE](#) function returns a copy of its string argument converted to lowercase. Similarly, the [STRUPCASE](#) function converts its argument to uppercase.

## White Space Removal

The [STRCOMPRESS](#) and [STRTRIM](#) functions can be used to eliminate unwanted white space (blanks or tabs) from their string arguments.

## Length

The [STRLEN](#) function returns the length of its string argument.

## Substrings

The [STRPOS](#), [STRPUT](#), and [STRMID](#) routines locate, insert, and extract substrings from their string arguments.

## Splitting and Joining Strings

The [STRSPLIT](#) function is used to break strings apart, and the [STRJOIN](#) function can be used to and glue strings together.

## Comparing Strings

The [STRCMP](#), [STRMATCH](#), and [STREGEX](#) functions perform string comparisons.

## Non-string and Non-scalar Arguments

Most of the string processing routines described in this chapter expect at least one argument that is the string on which they act.

If the argument is not of type string, IDL converts it to type string using the same default formatting rules that are used by the [PRINT/PRINTF](#) or [STRING](#) routines. The function then operates on the converted result. Thus, the IDL statement,

```
PRINT, STRLEN(23)
```

returns the result

```
8
```

because the argument “23” is first converted to the string ' 23' that happens to be a string of length 8.

If the argument is an array instead of a scalar, the function returns an array result with the same structure as the argument. Each element of the result corresponds to an element of the argument. For example, the following statements:

```
;Get an uppercase version of TREES.
A = STRUPCASE(trees)

;Show that the result is also an array.
HELP, A

;Display the original.
PRINT, trees

;Display the result.
PRINT, A
```

produce the following output:

```
A                STRING      = Array(7)
Beech Birch Mahogany Maple Oak Pine Walnut
BEECH BIRCH MAHOGANY MAPLE OAK PINE WALNUT
```

For more details on how individual routines handle their arguments, see the individual descriptions in the *IDL Reference Guide*.

# String Concatenation

The addition operator is used to concatenate strings. For example, the command:

```
A = 'This is' + ' a concatenation example.'  
PRINT, A
```

results in the following output:

```
This is a concatenation example.
```

The following IDL statements build a scalar string containing a comma-separated list of the names found in the TREES string array:

```
;Use REPLICATE to make an array with the correct number of commas  
;and add it to trees.  
names = trees + [REPLICATE(',', N_ELEMENTS(trees)-1), '']  
  
;Show the resulting list.  
PRINT, names
```

Running the above statements results in the following output:

```
Beech, Birch, Mahogany, Maple, Oak, Pine, Walnut
```

## Using STRING to Format Data

The STRING function has the following form:

$$S = \text{STRING}(\text{Expression}_1, \dots, \text{Expression}_n)$$

It converts its parameters to characters, returning the result as a string expression. It is identical in function to the PRINT procedure, except that its output is placed into a string rather than being output to the terminal. As with PRINT, the FORMAT keyword can be used to explicitly specify the desired format. See the discussions of free format and explicitly formatted input/output (“Free Format I/O” on page 356) for details of data formatting. For more information on the STRING function, see [STRING](#) in the *IDL Reference Guide*.

As a simple example, the following IDL statements:

```
;Produce a string array.
A = STRING(FORMAT='("The values are:", /, (I))', INDGEN(5))

;Show its structure.
HELP, A

;Print the result.
FOR I = 0, 4 DO PRINT, A[I]
```

produce the following output:

```
A  STRING      = Array(6)
The values are:
    0
    1
    2
    3
```

## Reading Data from Strings

The READS procedure performs formatted input from a string variable and writes the results into one or more output variables. This procedure differs from the READ procedure only in that the input comes from memory instead of a file.

This routine is useful when you need to examine the format of a data file before reading the information it contains. Each line of the file can be read into a string using READF. Then the components of that line can be read into variables using READS.

See the description of [READS](#) in the *IDL Reference Guide* for more details.



## Byte Arguments and Strings

There is a close association between a string and a byte array—a string is simply an array of bytes that is treated as a series of ASCII characters. Therefore, it is convenient to be able to convert between them easily.

When `STRING` is called with a single argument of byte type and the `FORMAT` keyword is not used, `STRING` does not work in its normal fashion. Instead of formatting the byte data and placing it into a string, it returns a string containing the byte values from the original argument. Thus, the result has one less dimension than the original argument. A two-dimensional byte array becomes a vector of strings, and a byte vector becomes a scalar string. However, a byte scalar also becomes a string scalar. For example, the statement

```
PRINT, STRING([72B, 101B, 108B, 108B, 111B])
```

produces the output below:

```
Hello
```

This output results because the argument to `STRING`, as produced by the array concatenation operator, is a byte vector. Its first element is `72B` which is the ASCII code for “H,” the second is `101B` which is an ASCII “e,” and so forth. Set the `PRINT` keyword can be used to disable this feature and cause `STRING` to treat byte data in the usual way.

As discussed in [Chapter 16, “Files and Input/Output”](#), it is easier to read fixed-length string data from binary files into byte variables instead of string variables. Therefore, it is convenient to read the data into a byte array and use this special behavior of `STRING` to convert the data into string form.

Another use for this feature is to build strings that contain nonprintable characters in a way such that the character is not entered directly. This results in programs that are easier to read and that also avoid file transfer difficulties (some forms of file transfer have problems transferring nonprintable characters). Due to the way in which strings are implemented in IDL, applying the `STRING` function to a byte array containing a null (zero) value will result in the resulting string being truncated at that position. Thus, the statement,

```
PRINT, STRING([65B, 66B, 0B, 67B])
```

produces the following output:

```
AB
```

This output is produced because the null byte in the third position of the byte array argument terminates the string and hides the last character.

**Note**

---

The `BYTE` function, when called with a single argument of type string, performs the inverse operation to that described above, resulting in a byte array containing the same byte values as its string argument. For additional information about the `BYTE` function, see [“Type Conversion Functions”](#) on page 110.

---

## Case Folding

The `STRLOWCASE` and `STRUPCASE` functions are used to convert arguments to lowercase or uppercase. They have the form:

```
S = STRLOWCASE(String)
```

```
S = STRUPCASE(String)
```

where *String* is the string to be converted to lowercase or uppercase.

The following IDL statements generate a table of the contents of `TREES` showing each name in its actual case, lowercase and uppercase:

```
FOR I=0, 6 DO PRINT, trees[I], STRLOWCASE(trees[I]), $
STRUPCASE(trees[I]), FORMAT = '(A, T15, A, T30, A)'
```

The resulting output from running this statement is as follows:

Beech	beech	BEECH
Birch	birch	BIRCH
Mahogany	mahogany	MAHOGANY
Maple	maple	MAPLE
Oak	oak	OAK
Pine	pine	PINE
Walnut	walnut	WALNUT

A common use for case folding occurs when writing IDL procedures that require input from the user. By folding the case of the response, it is possible to handle responses written in uppercase, lowercase, or mixed case. For example, the following IDL statements can be used to ask “yes or no” style questions:

```
;Create a string variable to hold the response.
answer = ''

;Ask the question.
READ, 'Answer yes or no: ', answer
IF (STRUPCASE(answer) EQ 'YES') THEN $
    ;Compare the response to the expected answer.
    PRINT, 'YES' ELSE PRINT, 'NO'
```

# Whitespace

The `STRCOMPRESS` and `STRTRIM` functions are used to remove unwanted white space (tabs and spaces) from a string. This can be useful when reading string data from arbitrarily formatted strings.

## Removing All Whitespace

The function `STRCOMPRESS` returns a copy of its string argument with all white space replaced with a single space or completely removed. It has the form:

```
S = STRCOMPRESS(String)
```

where *String* is the string to be compressed.

The default action is to replace each section of white space with a single space. Setting the `REMOVE_ALL` keyword causes white space to be completely eliminated. For example,

```
;Create a string with undesirable white space. Such a string might
;be the result of reading user input with a READ statement.
A = '   This   is a poorly   spaced   sentence.   '

;Print the result of shrinking all white space to a single blank.
PRINT, '>', STRCOMPRESS(A), '<'

;Print the result of removing all white space.
PRINT '>', STRCOMPRESS(A, /REMOVE_ALL), '<'
```

results in the output:

```
> This is a poorly spaced sentence. <
>Thisisapoorlyspacedsentence.<
```

## Removing Leading or Trailing Blanks

The function `STRTRIM` returns a copy of its string argument with leading and/or trailing white space removed. It has the form:

```
S = STRTRIM(String[, Flag])
```

where *String* is the string to be trimmed and *Flag* is an integer that indicates the specific trimming to be done. If *Flag* is 0 or is not present, trailing white space is removed. If it is 1, leading white space is removed. Both trailing and leading white space are removed if *Flag* is equal to 2. For example:

```
;Create a string with unwanted leading and trailing blanks.
```

```
A = ' This string has leading and trailing white space '
```

```
;Remove trailing white space.  
PRINT, '>', STRTRIM(A), '<'
```

```
;Remove leading white space.  
PRINT, '>', STRTRIM(A,1), '<'
```

```
;Remove both.  
PRINT, '>', STRTRIM(A,2), '<'
```

Executing these statements produces the output below.

```
> This string has leading and trailing white space<  
>This string has leading and trailing white space <  
>This string has leading and trailing white space<
```

## Removing All Types of Whitespace

When processing string data, `STRCOMPRESS` and `STRTRIM` can be combined to remove leading and trailing white space and shrink any white space in the middle down to single spaces.

```
;Create a string with undesirable white space.  
A = 'Yet  another poorly spaced  sentence.  '
```

```
;Eliminate unwanted white space.  
PRINT, '>' STRCOMPRESS(STRTRIM(A,2)), '<'
```

Executing these statements gives the result below:

```
>Yet another poorly spaced sentence.<
```

## Finding the Length of a String

The `STRLEN` function is used to obtain the length of a string. It has the form:

$$L = \text{STRLEN}(\textit{String})$$

where *String* is the string for which the length is required. For example, the following statement

```
PRINT, STRLEN('This sentence has 31 characters')
```

results in the output

```
31
```

while the following IDL statement prints the lengths of all the names contained in the array `TREES`.

```
PRINT, STRLEN(trees)
```

The resulting output is as follows:

```
5      5      8      5      3      4      6
```

# Substrings

IDL provides the [STRPOS](#), [STRPUT](#), and [STRMID](#) routines to locate, insert, and extract substrings from their string arguments.

## Searching for a Substring

The [STRPOS](#) function is used to search for the first occurrence of a substring. It has the form

$$S = \text{STRPOS}(\textit{Object}, \textit{Search\_string}[, \textit{Position}])$$

where *Object* is the string to be searched, *Search\_string* is the substring to search for, and *Position* is the character position (starting with position 0) at which the search is begun. If the optional argument *Position* is omitted, the search is started at the first character (character position 0). The following IDL procedure counts the number of times that the word “dog” appears in the string “dog cat duck rabbit dog cat dog”:

```

PRO Animals

;The search string, "dog", appears three times.
animals = 'dog cat duck rabbit dog cat dog'

;Start searching in character position 0.
I = 0

;Number of occurrences found.
cnt = 0

;Search for an occurrence.
WHILE (I NE -1) DO BEGIN
    I = STRPOS(animals, 'dog', I)

    IF (I NE -1) THEN BEGIN
        ;Update counter.
        cnt = cnt + 1

        ;Increment I so as not to count the same instance of 'dog'
        ;twice.
        I = I + 1
    ENDIF
ENDWHILE

;Print the result.
PRINT, 'Found ', cnt, " occurrences of 'dog'"

```

```
END
```

Running the above program produces the result below.

```
Found          3 occurrences of 'dog'
```

## Searching For the Last Occurrence of a Substring

The `REVERSE_SEARCH` keyword to the `STRPOS` function makes it easy to find the last occurrence of a substring within a string. In the following example, we search for the last occurrence of the letter “I” (or “i”) in a sentence:

```
sentence = 'IDL is fun.'
sentence = STRUPCASE(sentence)
lasti = STRPOS(sentence, 'I', /REVERSE_SEARCH)
PRINT, lasti
```

This results in:

```
4
```

Note that although `REVERSE_SEARCH` tells `STRPOS` to begin searching from the end of the string, the `STRPOS` function still returns the position of the search string starting from the beginning of the string (where 0 is the position of the first character).

## Inserting the Contents of One String into Another

The `STRPUT` procedure is used to insert the contents of one string into another. It has the form,

```
STRPUT, Destination, Source[, Position]
```

where *Destination* is the string to be overwritten, *Source* is the string to be inserted, and *Position* is the first character position within *Destination* at which *Source* will be inserted. If the optional argument *Position* is omitted, the overwrite is started at the first character (character position 0). The following IDL statements use `STRPOS` and `STRPUT` to replace every occurrence of the word “dog” with the word “CAT” in the string “dog cat duck rabbit dog cat dog”:

```
animals = 'dog cat duck rabbit dog cat dog'
;The string to search, "dog", appears three times.

;While any occurrence of "dog" exists, replace it.
WHILE (((I = STRPOS(animals, 'dog')) NE -1) DO $
STRPUT, animals, 'CAT', I

;Show the resulting string.
```



```
PRINT, animals
```

Running the above statements produces the result below.

```
CAT cat duck rabbit CAT cat CAT
```

## Extracting Substrings

The **STRMID** function is used for extracting substrings from a larger string. It has the form:

```
STRMID(Expression, First_Character [, Length])
```

where *Expression* is the string from which the substring will be extracted, *First\_Character* is the starting position within *Expression* of the substring (the first position is position 0), and *Length* is the length of the substring to extract. If there are not *Length* characters following the position *First\_Character*, the substring will be truncated. If the *Length* argument is not supplied, **STRMID** extracts all characters from the specified starting position to the end of the string. The following IDL statements use **STRMID** to print a table matching the number of each month with its three-letter abbreviation:

```
;String containing all the month names.
months = 'JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC'

;Extract each name in turn. The equation (I-1)*3 calculates the
;position within MONTH for each abbreviation
FOR I = 1, 12 DO PRINT, I, '      ', $
STRMID(months, (I - 1) * 3, 3)
```

The result of executing these statements is as follows:

```
1      JAN
2      FEB
3      MAR
4      APR
5      MAY
6      JUN
7      JUL
8      AUG
9      SEP
10     OCT
11     NOV
12     DEC
```

# Splitting and Joining Strings

The `STRSPLIT` function is used to break apart a string, and the `STRJOIN` function is used to glue together separate strings into a single string.

The `STRSPLIT` function uses the following syntax:

```
Result = STRSPLIT( String [, Pattern] )
```

where *String* is the string to be split, and *Pattern* is either a string of character codes used to specify the delimiter, or a regular expression, as implemented by the `STREGEX` function.

The `STRJOIN` function uses the following syntax:

```
Result = STRJOIN( String [, Delimiter] )
```

where *String* is the string or string array to be joined, and *Delimiter* is the separator string to use between the joined strings.

The following example uses `STRSPLIT` to extract words from a sentence into an array, modifies the array, and uses `STRJOIN` to rejoin the individual array elements into a new sentence:

```
str1 = 'Hello Cruel World'
words = STRSPLIT(str1, ' ', /EXTRACT)
newwords=[words[0],words[2]]
PRINT, STRJOIN(newwords, ' ')
```

This code results in the following output:

```
Hello World
```

In this example, the `EXTRACT` keyword caused `STRSPLIT` to return the substrings as array elements, rather than the default action of returning an array of character offsets indicating the position of each substring.

The `STRJOIN` function allows us to specify the delimiter used to join the strings. Instead of using a space as in the above example, we could use a different delimiter as follows:

```
str1 = 'Hello Cruel World'
words = STRSPLIT(str1, ' ', /EXTRACT)
newwords=[words[0],words[2]]
PRINT, STRJOIN(newwords, ' Kind ')
```

This code results in the following output:

```
Hello Kind World
```

# Comparing Strings

IDL provides several different mechanisms for performing string comparisons. In addition to the EQ operator, the STRCMP, STRMATCH, and STREGEX functions can all be used for string comparisons.

## Case-Insensitive Comparisons of the First N Characters

The **STRCMP** function simplifies case-insensitive comparisons, and comparisons of only the first *N* characters of two strings. The STRCMP function uses the following syntax:

$$Result = STRCMP( String1, String2 [, N] )$$

where *String1* and *String2* are the strings to be compared, and *N* is the number of characters from the beginning of the string to compare.

Using the EQ operator to compare the first 3 characters of the strings “Moose” and “mOO” requires the following steps:

```
A = 'Moose'
B = 'mOO'
```

```
C=STRMID(A,0,3)
```

```
IF (STRLOWCASE(C) EQ STRLOWCASE(B)) THEN PRINT, "It's a match!"
```

Using the EQ operator for this case-insensitive comparison of the first 3 characters requires the STRMID function to extract the first 3 characters, and the STRLOWCASE (or STRUPCASE) function to change the case.

The STRCMP function could be used to simplify this comparison:

```
A='Moose'
B='mOO'
```

```
IF (STRCMP(A,B,3, /FOLD_CASE) EQ 1) THEN PRINT, "It's a match!"
```

The optional *N* argument of the STRCMP function allows us to easily specify how many characters to compare (from the beginning of the input strings), and the FOLD\_CASE keyword specifies a case-insensitive search. If *N* is omitted, the full strings are compared.

## String Comparisons Using Wildcards

The `STRMATCH` function can be used to compare a search string containing wildcard characters to another string. It is similar in function to the way the standard UNIX command shell processes file wildcard characters.

The `STRMATCH` function uses the following syntax:

```
Result = STRMATCH( String, SearchString )
```

where *String* is the string in which to search for *SearchString*.

*SearchString* can contain the following wildcard characters:

Wildcard Character	Description
*	Matches any string, including the null string.
?	Matches any single character.
[...]	Matches any one of the enclosed characters. A pair of characters separated by "-" matches any character lexically between the pair, inclusive. If the first character following the opening [ is a !, any character not enclosed is matched. To prevent one of these characters from acting as a wildcard, it can be quoted by preceding it with a backslash character (e.g. "\*" matches the asterisk character). Quoting any other character (including \ itself) is equivalent to the character (e.g. "\a" is the same as "a").

Table 9-1: Wildcard Characters used by `STRMATCH`

The following examples demonstrate various uses of wildcard matching:

**Example 1:** Find all 4-letter words in a string array that begin with “f” or “F” and end with “t” or “T”:

```
str = ['foot', 'Feet', 'fate', 'FAST', 'ferret', 'fort']
PRINT, str[WHERE(STRMATCH(str, 'f??t', /FOLD_CASE) EQ 1)]
```

This results in:

```
foot Feet FAST fort
```

**Example 2:** Find words of any length that begin with “f” and end with “t”:

```
str = ['foot', 'Feet', 'fate', 'FAST', 'ferret', 'fort']
PRINT, str[WHERE(STRMATCH(str, 'f*t', /FOLD_CASE) EQ 1)]
```

This results in:

```
foot Feet FAST ferret fort
```

**Example 3:** Find 4-letter words beginning with “f” and ending with “t”, with any combination of “o” and “e” in between:

```
str = ['foot', 'Feet', 'fate', 'FAST', 'ferret', 'fort']
PRINT, str[WHERE(STRMATCH(str, 'f[eo][eo]t', /FOLD_CASE) EQ 1)]
```

This results in:

```
foot Feet
```

**Example 4:** Find all words beginning with “f” and ending with “t” whose second character is not the letter “o”:

```
str = ['foot', 'Feet', 'fate', 'FAST', 'ferret', 'fort']
PRINT, str[WHERE(STRMATCH(str, 'f[!o]*t', /FOLD_CASE) EQ 1)]
```

This results in:

```
Feet FAST ferret
```

## Complex Comparisons Using Regular Expressions

A more difficult search than the one above would be to find words of any length beginning with “f” and ending with “t” without the letter “o” in between. This would be difficult to accomplish with STRMATCH, but could be easily accomplished using the [STREGEX](#) function:

```
str = ['foot', 'Feet', 'fate', 'FAST', 'ferret', 'fort']
PRINT, STREGEX(str, '^f[^o]*t$', /EXTRACT, /FOLD_CASE)
```

This statement results in:

```
Feet FAST ferret
```

Note the following about this example:

- Unlike the \* wildcard character used by STRMATCH, the \* meta character used by STREGEX applies to the item directly on its left, which in this case is [^o], meaning “any character except the letter ‘o’”. Therefore, [^o]\* means “zero or more characters that are not ‘o’”, whereas the following statement would find only words whose second character is not “o”:

```
PRINT, str[WHERE(STRMATCH(str, 'f[!o]*t', /FOLD_CASE) EQ 1)]
```

- The anchors (^ and \$) tell STREGEX to find only words that begin with “f” and end with “t”. If we left out the \$ anchor, STREGEX would also return “fat”, which is a substring of “fate”.

Regular expressions are somewhat more difficult to use than simple wildcard matching (which is why the UNIX shell does matching) but in exchange offers unparalleled expressive power.

For more on the STREGEX function, see [STREGEX](#) in the *IDL Reference Guide*, and for an introduction to regular expressions, see “[Learning About Regular Expressions](#)” on page 191.

# Learning About Regular Expressions

Regular expressions are a very powerful way to match arbitrary text. Stemming from neurophysiological research conducted in the early 1940's, their mathematical foundation was established during the 1950's and 1960's. Their use has a long history in computer science, and they are an integral part of many UNIX tools, including `awk`, `egrep`, `lex`, `perl`, and `sed`, as well as many text editors. Regular expressions are slower than simple pattern matching algorithms, and they can be cryptic and difficult to write correctly. Small mistakes in specification can yield surprising results. They are, however, vastly more succinct and powerful than simple pattern matching, and can easily handle tasks that would be difficult or impossible otherwise.

The topic of regular expressions is a very large one, complicated by the arbitrary differences in the implementations found in various tools. Anything beyond an extremely simplistic sketch is well beyond the scope of this manual. To understand them better, we recommend a good text on the subject, such as “Mastering Regular Expressions”, by Jeffrey E.F. Friedl (O'Reilly & Associates, Inc, ISBN 1-56592-257-3). The following is an abbreviated, simplified, and incomplete explanation of regular expressions, sufficient to gain a cursory understanding of them.

The regular expression engine attempts to match the regular expression against the input string. Such matching starts at the beginning of the string and moves from left to right. The matching is considered to be “greedy”, because at any given point, it will always match the longest possible substring. For example, if a regular expression could match the substring ‘aa’ or ‘aaa’, it will always take the longer option.

## Meta Characters

A regular expression “ordinary character” is a character that matches itself. Most characters are ordinary. The exceptions, sometimes called “meta characters”, have special meanings. To convert a meta character into an ordinary one, you “escape” it by preceding it with a backslash character (e.g. `\*`). The meta characters are described in the following table:

Character	Description
.	The period matches any character.

*Table 9-2: Meta characters*

Character	Description
[ ]	The open bracket character indicates a “bracket expression”, which is discussed below. The close bracket character terminates such an expression.
\	The backslash suppresses the special meaning of the character it precedes, and turns it into an ordinary character. To insert a backslash into your regular expression pattern, use a double backslash (“\\”).
( )	The open parenthesis indicates a “subexpression”, discussed below. The close parenthesis character terminates such a subexpression.
<b>Repetition Characters</b>	These characters are used to specify repetition. The repetition is applied to the character or expression directly to the left of the repetition operator.
*	Zero or more of the character or expression to the left. Hence, 'a*' means “zero or more instances of 'a'”.
+	One or more of the character or expression to the left. Hence, 'a+' means “one or more instances of 'a'”.
?	Zero or one of the character or expression to the left. Hence, 'a?' will match 'a' or the empty string “”.
{ }	An interval qualifier allows you to specify exactly how many instances of the character or expression to the left to match. If it encloses a single unsigned integer length, it means to match exactly that number of instances. Hence, 'a{3}' will match 'aaa'. If it encloses 2 such integers separated by a comma, it specifies a range of possible repetitions. For example, 'a{2,4}' will match 'aa', 'aaa', or 'aaaa'. Note that '{0,1}' is equivalent to '?'.
	Alternation. This operator is used to indicate that one of several possible choices can match. For example, '(a b c)z' will match any of 'az', 'bz', or 'cz'.

Table 9-2: Meta characters



Character	Description
^ \$	Anchors. A '^' matches the beginning of a string, and '\$' matches the end. As we have seen above, regular expressions usually match any possible substring. Anchors can be used to change this and require a match to occur at the beginning or end of the string. For example, '^abc' will only match strings that start with the string 'abc'. 'abc\$' will only match a string containing <i>only</i> 'abc'.

Table 9-2: Meta characters

## Subexpressions

Subexpressions are those parts of a regular expression enclosed in parentheses. There are two reasons to use subexpressions:

- To apply a repetition operator to more than one character. For example, '(fun){3}' matches 'funfunfun', while 'fun{3}' matches 'funnn'.
- To allow location of the subexpression using the SUBEXPR keyword to STREGEX.

## Bracket Expressions

Bracket expressions (expressions enclosed in square brackets) are used to specify a set of characters that can satisfy a match. Many of the meta characters described above (.\*[\]) lose their special meaning within a bracket expression. The right bracket loses its special meaning if it occurs as the first character in the expression (after an initial '^', if any).

There are several different forms of bracket expressions, including:

- **Matching List** — A matching list expression specifies a list that matches any one of the characters in the list. For example, '[abc]' matches any of the characters 'a', 'b', or 'c'.
- **Non-Matching List** — A non-matching list expression begins with a '^', and specifies a list that matches any character *not* in the list. For example, '[^abc]' matches any characters *except* 'a', 'b', or 'c'. The '^' only has this special meaning when it occurs first in the list immediately after the opening '['.
- **Range Expression** — A range expression consists of 2 characters separated by a hyphen, and matches any characters lexically within the range indicated. For

example, '[A-Za-z]' will match any alphabetic character, upper or lower case. Another way to get this effect is to specify '[a-z]' and use the `FOLD_CASE` keyword to `STREGEX`.



# Chapter 10: Statements

The following topics are covered in this chapter:

---

Overview .....	196	Function Definition Statement .....	216
Components of Statements .....	197	GOTO Statement .....	219
The Assignment Statement .....	198	IF Statement .....	220
Blocks .....	204	Procedure Call Statement .....	222
CASE Statement .....	206	Procedure Definition Statement .....	225
Common Blocks .....	208	REPEAT Statement .....	226
FOR Statement .....	211	WHILE Statement .....	227

# Overview

IDL programs, procedures, and functions are composed of one or more valid statements. Most simple IDL statements can also be entered in the immediate mode in response to the IDL> prompt. The thirteen types of IDL statements are as follows:

- Assignment
- Block
- Case
- Common Block Definition
- For
- Forward Function Definition
- Function Definition
- Goto
- If
- Procedure Call
- Procedure Definition
- Repeat
- While

# Components of Statements

Statements in IDL can consist of any combination of three parts:

- A label field
- The statement proper
- A comment field

Spaces and tabs can appear anywhere except in the middle of an identifier or numeric constant.

## Statement Labels

Labels are the destinations of GOTO statements as well as the ON\_ERROR and ON\_IOERROR procedures. The label field, which must appear before the statement or comment, is simply an identifier followed by a colon. A line can consist of only a label field. Label identifiers, as with variable names, consist of 1 to 15 alphanumeric characters and are case insensitive. The dollar sign (\$) and underscore (\_) characters can appear after the first character. Some examples of labels are as follows:

```
LABEL1:  
LOOP_BACK: A = 12  
I$QUIT: RETURN ;Comments are allowed.
```

## Comments

The comment field, which is ignored by IDL, begins with a semicolon and continues to the end of the line. Lines can consist of only a comment field or can be entirely blank. It is good programming practice to fully annotate programs with comments. There are no execution-time or space penalties for comments in IDL.

The following IDL statement shows a simple assignment statement with a comment field:

```
COUNT = 5 ;Set variable COUNT to 5. This is the comment field.
```

# The Assignment Statement

The assignment statement stores a value in a variable. There are four forms of the assignment statement, as shown in the following table.

Syntax	Subscript Structure	Expression Structure	Effect
<i>Variable = Expression</i>	<i>None</i>	All	<i>Expression is stored in Variable</i>
<i>Variable[Subscripts] = Expression</i>	Scalar	Scalar	<i>Expression is stored in a single element of Variable</i>
	Scalar	Array	<i>Expression array is inserted in Variable array</i>
	Array	Scalar	<i>Expression scalar is stored in designated elements of Variable</i>
	Array	Array	<i>Elements of Expression are stored in designated elements of Variable</i>
<i>Variable[Range] = Expression</i>	<i>Range</i>	Scalar	Scalar is inserted into subarray
<i>Variable[Range] = Expression</i>	<i>Range</i>	Array	Illegal

Table 10-1: Types of Assignment Statements

In the second and fourth cases, an array can be used as a subscript. This stores the values from the right side of the statement into elements of the variable designated by the contents of the array subscript.

---

**Note**

In versions of IDL prior to version 5.0, parentheses were used to enclose array subscripts. While using parentheses to enclose array subscripts will continue to work as in previous version of IDL, we strongly suggest that you use brackets in all new code. See “[Array Subscript Syntax: \[ \] vs. \( \)](#)” on page 157 for additional details.

---

## The Basic Assignment Statement

The first, and most basic, form of the assignment statement is as follows:

$$\text{Variable} = \text{Expression}$$

The old value of the variable, if any, is discarded, and the value of the expression is stored in the variable. The expression on the right side can be of any type or structure.

### Examples

Some examples of the basic form of the assignment statement are as follows:

```
;Set mmax to value.
mmax = 100 * X + 2.987

;name becomes a scalar string variable.
name = 'Mary'

;Make arr a 100-element, floating-point array.
arr = FLTARR(100)

;Discard points 0 to 49 of arr. It is now a 50-element array.
arr = arr[50:*]
```

## The Second Form of the Assignment Statement

The second type of assignment statement has the following form:

$$\text{Variable}[\text{Subscripts}] = \text{Scalar\_Expression}$$

Here, a single element of the specified array is set to the value of the scalar expression. The expression can be of any type and is converted, if necessary, to the type of the variable. The variable on the left side must be either an array or a file

variable. Some examples of assigning scalar expressions to subscripted variables are shown below:

```
;Set element 100 of data to value.
data[100] = 1.234999

;Store string in an array. name must be a string array or an error
;will result.
name[index] = 'Joe'

;Set element [X, Y] of the 2-dimensional array image to the value
contained in pixel.
image[X, Y] = pixel
```

## Using Array Subscripts with the Second Form of the Assignment Statement

The subscripted variable can have either a scalar or array subscript. If the subscript expression is an array, the scalar value is stored in the elements of the array whose subscripts are elements of the subscript array. For example, the statement

```
data[[3, 5, 7, 9]] = 0
```

zeroes the four specified elements of `data`: `data[3]`, `data[5]`, `data[7]` and `data[9]`.

The subscript array is converted to longword type if necessary before use. Elements of the subscript array that are negative, or greater than the highest subscript of the subscripted array, are clipped to the target array boundaries. Note that a common error is to use a negative *scalar* subscript (e.g., `A[-1]`). Using this type of subscript causes an error. Negative *array* subscripts (e.g., `A[[-1]]`) do not cause errors.

The [WHERE](#) function can be used to select array elements to be changed. For example, the statement:

```
data[WHERE(data LT 0)] = -1
```

sets all negative elements of `data` to -1 without changing the positive elements. The result of the function, `WHERE(data LT 0)`, is a vector composed of the subscripts of the negative elements of `data`. Using this vector as a subscript changes only the negative elements.

## The Third Form of the Assignment Statement

The third type of assignment statement is similar to the second, except the subscripts specify a range in which all elements are set to the scalar expression.

```
Variable[Subscript_Range] = Scalar_Expression
```



A subscript range specifies a beginning and ending subscript. The beginning and ending subscripts are separated by the colon character. An ending subscript equal to the size of the dimension minus one can be written as `*`.

For example, `arr[I:J]` denotes those points in the vector `arr` with subscripts between `I` and `J` inclusive. `I` must be less than or equal to `J` and greater than or equal to zero. `J` denotes the points in `arr` from `arr[I]` to the last point and must be less than the size of the dimension `arr [I:*`]. See [Chapter 8, “Array Subscripts”](#) for more details on subscript ranges.

## Examples

Assuming the variable `B` is a  $512 \times 512$ -byte array, some examples are as follows:

```
;Store 1 in every element of the i-th row.
array[* , I] = 1

;Store 1 in every element of the j-th column.
array[J , *] = 1

;Zero all the rows of columns 200 through 220 of array.
array[200:220 , *] = 0

;Store the value 100 in all the elements of array.
array[*] = 100
```

## The Fourth Form of the Assignment Statement

The fourth type of assignment statement is of the following form:

$$\text{Variable}[\text{Subscripts}] = \text{Array}$$

Note that this form is syntactically identical to the second type of assignment statement, but that the expression on the right-hand-side is an array instead of a scalar. This form of the assignment statement is used to insert one array into another.

The array expression on the right is inserted into the array appearing on the left side of the equal sign starting at the point designated by the subscripts.

## Examples

For example, to insert the contents of an array called `A` into array `B`, starting at point `B[13, 24]`, use the following statement:

```
B[13 , 24] = A
```

If `A` is a 5-column by 6-row array, elements `B[13:17, 24:29]` are replaced by the contents of array `A`.

In the next example, a subarray is moved from one position to another with the statement:

```
B[100, 200] = B[200:300, 300:400]
```

A subarray of B, specifically the columns 200 to 300 and rows 300 to 400, is moved to columns 100 to 200 and rows 200 to 300, respectively.

## Using Array Subscripts with the Fourth Form of the Assignment Statement

If the subscript expression applied to the variable is an array and an array appears on the right side of the statement:

```
Variable[Array] = Array
```

then elements from the right side are stored in the elements designated by the subscript vector. Only those elements of the subscripted variable whose subscripts appear in the subscript vector are changed. For example, the statement

```
B[[ 2, 4, 6 ]] = [4, 16, 36]
```

is equivalent to the following series of assignment statements:

```
B[2] = 4
B[4] = 16
B[6] = 36
```

Subscript elements are interpreted as if the subscripted variable is a vector. For example, if A is a  $10 \times n$  matrix, the element  $A[i, j]$  has the subscript  $i+10*j$ . The subscript array is converted to longword type before use, if necessary.

As described previously for the second form of assignment statement, elements of the subscript array that are negative or larger than the highest subscript are clipped to the target array boundaries. Note that a common error is to use a negative *scalar* subscript (e.g., A[-1]). Using this type of subscript causes an error. Negative *array* subscripts (e.g., A[[-1]]) do not cause errors.

As another example, assume that the vector DATA contains data elements and that a data drop-out is denoted by a negative value. In addition, assume that there are never two or more adjacent drop-outs. The following statements replace all drop-outs with the average of the two adjacent good points:

```
;Subscript vector of drop-outs.
bad = WHERE(data LT 0)

;Replace drop-outs with average of previous and next point.
data[bad] = (data[bad - 1] + data[bad + 1]) / 2
```

In this example, the following actions are performed:

- We use the LT (less than) operator to create an array, with the same dimensions as `data`, that contains a 1 for every element of `data` that is less than zero and a zero for every element of `data` that is zero or greater. We use this “drop-out array” as a parameter for the `WHERE` function, which generates a vector that contains the one-dimensional *subscripts* of the elements of the drop-out array that are nonzero. The resulting vector, stored in the variable `bad`, contains the subscripts of the elements of `data` that are less than zero.
- The expression `data[bad - 1]` is a vector that contains the subscripts of the points immediately preceding the drop-outs; while similarly, the expression `data[bad + 1]` is a vector containing the subscripts of the points immediately after the drop-outs.
- The average of these two vectors is stored in `data[bad]`, the points that originally contained drop-outs.

## Associated Variables in Assignment Statements

A special case occurs when using an associated file variable in an assignment statement. For additional information regarding the `ASSOC` function, see [ASSOC](#) in the *IDL Reference Guide*. When a file variable is referenced, the last (and possibly only) subscript denotes the record number of the array within the file. This last subscript must be a simple subscript. Other subscripts and subscript ranges, except the last, have the same meaning as when used with normal array variables.

An implicit extraction of an element or subarray in a data record can also be performed. For example:

```

;Variable A associates the file open on unit 1 with the records of
;200-element, floating-point vectors.
A = ASSOC(1, FLTARR(200))

;Then, X is set to the first 100 points of record number 2, the
;third record of the file.
X = A[0:99, 2]

;Set the 24th point of record 16 to 12.
A[23, 16] = 12

;Increment points 10 to 199 of record 12. Points 0 to 9 of the
;record remain unchanged.
A[10, 12] = A[10:*, 12]+1

```

# Blocks

A block of statements is simply a group of statements that are treated as a single statement. For example, a group of statements can be bracketed as follows:

```
BEGIN
    Statement1
    ...
    Statementn
END
```

Blocks are necessary when more than one statement is the subject of a conditional or repetitive statement, as in the FOR, WHILE, and IF statements. In general, the format of a FOR statement with a block subject is as follows:

```
FOR Variable = Expression, Expression DO BEGIN
    Statement1
    Statement2
    ...
    ...
    ...
    Statementn
ENDFOR
```

All the statements between the BEGIN and the END are the subject of the FOR statement. The group of statements is executed as a single statement and is called a *compound statement*. Blocks can be nested within other blocks.

Syntactically, a block of statements is composed of one or more statements of any type, started by a BEGIN identifier and ended by an END identifier. IDL allows the use of blocks wherever a single statement is allowed. As an example, the process of reversing an array in place might be written as follows:

```
FOR I = 0, (N - 1)/2 DO BEGIN
    T = arr[I]
    arr[I] = arr[N - I - 1]
    arr[N - I - 1] = T
ENDFOR
```

---

## Note

The code shown above is for illustration only. The IDL [REVERSE](#) function is much more efficient.

---

The three statements between the BEGIN and ENDFOR are the subject of the FOR statement, and each will be executed once during each iteration of the loop. If the statements are not enclosed in a block, only the first statement (`T = arr[I]`) is executed

during each iteration, and the remaining two statements are executed only once after the termination of the FOR statement.

To ensure proper nesting of compound statements (one or more different blocks), the “END” statement terminating the block can be followed by the block type as shown in the following table. The compiler checks the end of each block, comparing it with the type of the enclosing statement. Any block can be terminated by the generic END, although no type checking is performed.

<b>END Statement</b>	<b>Use</b>
ENDCASE	CASE <i>statement</i>
ENDELSE	IF <i>statement</i> , ELSE <i>clause</i>
ENDFOR	FOR <i>statement</i>
ENDIF	IF <i>statement</i> , THEN <i>clause</i>
ENDREP	REPEAT <i>statement</i>
ENDWHILE	WHILE <i>statement</i>

*Table 10-2: Types of END Statements*

Listings produced by the IDL compiler indent each block four spaces to the right of the previous level to make the program structure easier to read. (See [.RUN](#) in the *IDL Reference Guide* for details on producing program listings with the IDL compiler.)

# CASE Statement

The CASE statement is used to select one, and only one, statement for execution, depending upon the value of the expression following the word CASE. This expression is called the case selector expression. The general form of the CASE statement is as follows:

```
CASE Expression OF
    Expression: Statement
    ...
    Expression: Statement
ELSE: Statement
ENDCASE
```

Each statement that is part of a CASE statement is preceded by an expression that is compared to the value of the selector expression. If a match is found, the statement is executed and control resumes directly below the CASE statement.

The ELSE clause of the CASE statement is optional. If included, it must be the last clause in the case statement. The statement after the ELSE is executed only if none of the preceding statement expressions match. If the ELSE is not included and none of the values match, an error occurs and program execution stops.

## Example

An example of the CASE statement follows:

```
CASE name OF
    ;If name is "Linda," print "sister".
    'Linda': PRINT, 'sister'

    ;If name is "John," print "brother".
    'John': PRINT, 'brother'

    ;If name is "Harry," print "step-brother".
    'Harry': PRINT, 'step-brother'

    ;No matches, print "Not a sibling."
ELSE: PRINT, 'Not a sibling.'

;End of CASE statement.
ENDCASE
```

Another example shows the CASE statement with the number 1 as the selector expression of the CASE. One is equivalent to *true* and is matched against each of the conditionals.

```
CASE 1 OF
  (X GT 0) AND (X LE 50): Y = 12 * X + 5
  (X GT 50) AND (X LE 100): Y = 13 * X + 4
  (X LE 200): BEGIN
    Y = 14 * X - 5
    Z = X + Y
  END
ELSE: PRINT, 'X has an illegal value.'
ENDCASE
```

In this CASE statement, only one clause is selected, and that clause is the first one whose value is equal to the value of the case selector expression. Each clause is tested in order, so it is most efficient to order the most frequently selected clauses first.

# Common Blocks

Common blocks are useful when there are variables that need to be accessed by several IDL procedures or when the value of a variable within a procedure must be preserved across calls. Once a common block has been defined, any program unit referencing that common block can access variables in the block as though they were local variables. Variables in a common statement have a global scope within procedures defining the same common block. Unlike local variables, variables in common blocks are not destroyed when a procedure is exited.

There are two types of common block statements: definition statements and reference statements.

## Common Block Definition Statements

The common block definition statement creates a common block with the designated name and places the variables whose names follow into that block. Variables defined in a common block can be referenced by any program unit that declares that common block. The general form of the COMMON block definition statement is as follows:

```
COMMON Block_Name, Variable1, Variable2, ..., Variablen
```

The number of variables appearing in the common block definition statement determines the size of the common block. The first program unit (main program, function, or procedure) defining the common block sets the size of the common block, which can never be expanded. Other program units can reference the common block with any number of variables up to the number originally specified. Different program units can give the variables different names, as shown in the example below.

Common blocks share the same space for all procedures. In IDL, common block variables are matched variable to variable, unlike FORTRAN, where storage locations are matched. The third variable in a given IDL common block will always be the same as the third variable in all declarations of the common block regardless of the size, type, or structure of the preceding variables.

Note that common blocks must appear before any of the variables they define are referenced in the procedure.

Variables in common blocks can be of any type and can be used in the same manner as normal variables. Variables appearing as parameters cannot be used in common blocks. There are no restrictions on the number of common blocks used, although each common block uses dynamic memory.



## Example

The two procedures in the following example show how variables defined in common blocks are shared.

```

PRO ADD, A
  COMMON SHARE1, X, Y, Z, Q, R
  A = X + Y + Z + Q + R
  PRINT, X, Y, Z, Q, R, A
  RETURN
END

PRO SUB, T
  COMMON SHARE1, A, B, C, D
  T = A - B - C - D
  PRINT, A, B, C, D, T
  RETURN
END

```

The variables X, Y, Z, and Q in the procedure ADD are the same as the variables A, B, C, and D, respectively, in procedure SUB. The variable R in ADD is not used in SUB. If the procedure SUB were to be compiled before the procedure ADD, an error would occur when the COMMON definition in ADD was compiled. This is because SUB has already declared the size of the common block, SHARE1, which cannot be extended.

## Common Block Reference Statements

The common block reference statement duplicates the common block and variable names from a previous definition. The common block need only be defined in the first routine to be compiled that references the block.

## Example

The two procedures in the following example share the common block SHARE2 and all its variables.

```

PRO MULT, M
  COMMON SHARE2, E, F, G
  M = E * F * G
  PRINT, M, E, F, G
  RETURN
END

PRO DIV, D
  COMMON SHARE2
  D = E / F
  PRINT, D, E, F, G

```

```
    RETURN  
END
```

The `MULT` procedure uses a common block *definition* statement to define the block `SHARE2`. The `DIV` procedure then uses a common block *reference* statement to gain access to all the variables defined in `SHARE2`. (Note that `MULT` must be defined before `DIV` in order for the common block *reference* to succeed.)

# FOR Statement

The FOR statement is used to execute one or more statements repeatedly, while incrementing or decrementing a variable with each repetition, until a condition is met. It is analogous to the DO statement in FORTRAN.

In IDL, there are two types of FOR statements: one with an implicit increment of 1 and the other with an explicit increment. If the condition is not met the first time the FOR statement is executed, the subject statement is not executed.

## FOR Statement with an Increment of One

The FOR statement with an implicit increment of one is written as follows:

```
FOR Variable = Expression, Expression DO Statement
```

The variable after the FOR is called the index variable and is set to the value of the first expression. The subject statement is executed, and the index variable is incremented by 1 until the index variable is larger than the second expression. This second expression is called the limit expression. Complex limit and increment expressions are converted to floating-point type.

### Warning

---

The data type of the index variable is determined by the type of the initial value expression. Keep this fact in mind to avoid the following:

```
FOR I = 0, 50000 DO ... ..
```

This loop does not produce the intended result. Converting the longword constant 50,000 to a short integer yields -15,536 because of truncation. The loop is not executed. The index variable's initial value is larger than the limit variable. The loop should be written as follows:

```
FOR I = 0L, 50000 DO ... ..
```

Note also that changing the data type of an index variable within a loop is not allowed, and will cause an error.

---

### Warning

---

Also be aware of FOR loops that are entered but are not terminated after the expected number of iterations, because of the truncation effect. For example, if the

index value exceeds the maximum value for the initial data type (and so is truncated) when it is expected instead to exceed the specified index limit, then the loop will continue beyond the expected number of iterations.

The following FOR statement continues infinitely:

```
FOR i = 0B, 240, 16 DO PRINT, i
```

The problem occurs because the variable `i` is initialized to a byte type with `0B`. After the index reaches the limit value `240B`, `i` is incremented by `16`, causing the value to go to `256B`, which is interpreted by IDL as `0B`, because of the truncation effect. As a result, the FOR loop “wraps around” and the index can never be exceeded.

## Examples

A simple FOR statement:

```
FOR I = 1, 4 DO PRINT, I, I^2
```

which produces the following output:

```
1    1
2    4
3    9
4   16
```

The index variable `I` is first set to an integer variable with a value of one. The call to the `PRINT` procedure is executed, then the index is incremented by one. This is repeated until the value of `I` is greater than four at which point execution continues at the statement following the FOR statement.

The next example displays the use of a block structure (instead of a single statement) as the subject of the FOR statement. The example is a common process used for computing a count-density histogram. **Note:** A `HISTOGRAM` function is provided by IDL.

```
FOR K = 0, N - 1 DO BEGIN
    C = A[K]
    HIST(C) = HIST(C)+1
ENDFOR
```

The next example displays a FOR statement with floating-point index and limit expressions:

```
FOR X = 1.5, 10.5 DO S = S + SQRT(X)
```

where *X* is set to a floating-point variable and steps through the values (1.5, 2.5, ..., 10.5).

The indexing variables and expressions can be integer, longword, floating-point, or double-precision. The type of the index variable is determined by the type of the first expression after the “=” character.

### Warning

---

Due to the inexact nature of IEEE floating-point numbers, using floating-point indexing can cause “infinite loops” and other problems. This problem is also manifested in both the C and FORTRAN programming languages. For example, the numbers 0.1, 0.01, 1.6, and 1.7 do not have exact representations under the IEEE standard. To see this phenomenon, enter the following IDL command:

```
PRINT, 0.1, 0.01, 1.6, 1.7, FORMAT='(f20.10)'
```

IDL prints the following *approximations* to the numbers we requested:

```
0.1000000015  
0.0099999998  
1.6000000238  
1.7000000477
```

See [Accuracy & Floating-Point Operations](#) in the *Using IDL* manual for more information about floating-point numbers.

---

## FOR Statement with Variable Increment

The format of the second type of FOR statement is as follows:

```
FOR Variable = Expression1, Expression2, Increment DO Statement
```

This form is used when an increment other than 1 is desired.

The first two expressions describe the range of numbers for the index variable. The Increment specifies the increment of the index variable. A negative increment allows the index variable to step downward. In this case, the first expression must have a value greater than that of the second expression. If not, an almost infinite loop will result.

### Examples

The following examples demonstrate the second type of FOR statement.

```

;Decrement, K has the values 100., 99., ..., 1.
FOR K = 100.0, 1.0, -1 DO ...

;Increment by 2., loop has the values 0., 2., 4., ..., 1022.
FOR loop = 0, 1023, 2 DO ...

;Divide range from bottom to top by 4.
FOR mid = bottom, top, (top - bottom)/4.0 DO ...

```

### Warning

---

If the value of the increment expression is zero, an infinite loop occurs. A common mistake resulting in an infinite loop is a statement similar to the following:

```
FOR X = 0, 1, .1 DO ....
```

The variable `x` is first defined as an integer variable because the initial value expression is an integer zero constant. Then the limit and increment expressions are converted to the type of `x`, integer, yielding an increment value of zero because `.1` converted to integer type is 0. The correct form of the statement is:

```
FOR X = 0., 1, .1 DO ....
```

which defines `x` as a floating-point variable.

---

## Operation of the FOR Statement

The FOR statement performs the following steps:

1. The value of the first expression is evaluated and stored in the specified variable, which is called the index variable. The index variable is set to the type of this expression.
2. The value of the second expression is evaluated, converted to the type of the index variable, and saved in a temporary location. This value is called the limit value.
3. The value of the third expression, called the step value, is evaluated, type-converted if necessary, and stored. If omitted, a value of 1 is assumed.
4. If the index variable is greater than the limit value (in the case of a positive step value) the FOR statement is finished and control resumes at the next statement. Similarly, in the case of a negative step value, if the index variable is less than the limit value, control resumes after the FOR statement.
5. The statement or block following the DO is executed.

6. The step value is added to the index variable.
7. Steps 4, 5, and 6 are repeated until the test of Step 4 fails.

# Function Definition Statement

The syntax of the FUNCTION statement is as follows:

```
FUNCTION Function_Name, Parameter1, Parameter2, ..., Parametern
```

A function is a program unit containing one or more IDL statements that returns a value. This unit executes independently of its caller. It has its own local variables and execution environment. Once a function has been defined, references to the function cause the program unit to be executed. All functions return a function value which is given as a parameter in the RETURN statement used to exit the function. Function names can be up to 128 characters long.

The general format of a function definition is as follows:

```
FUNCTION Name, Parameter1, ..., Parametern
    Statement1
    Statement2
    ...
    ...
    RETURN, Expression
END
```

## Example

To define a function called AVERAGE, which returns the average value of an array, use the following statements:

```
FUNCTION AVERAGE, arr
    RETURN, TOTAL(arr)/N_ELEMENTS(arr)
END
```

Once the function AVERAGE has been defined, it is executed by entering the function name followed by its arguments enclosed in parentheses. Assuming the variable X contains an array, the statement,

```
PRINT, AVERAGE(X^2)
```

squares the array X, passes this result to the AVERAGE function, and prints the result. Parameters passed to functions are identified by their position or by a keyword. See “[Keyword Parameters](#)” on page 223. For further details about user-defined functions, see [Chapter 13, “Defining Procedures and Functions”](#).

## Automatic Execution

IDL automatically compiles and executes a user-written function or procedure when it is first referenced if:



1. The source code of the function is in the current working directory or in a directory in the IDL search path defined by the system variable `!PATH`.
2. The name of the file containing the function is the same as the function name suffixed by `.pro` or `.sav`. Under UNIX, the suffix should be in lowercase letters.

---

**Note**

IDL is case-insensitive. However, for some operating systems, IDL only checks for the lowercase filename based on the name of the procedure or function. We recommend that all filenames be named with lowercase.

---

---

**Warning**

User-written functions must be defined before they are referenced, unless they meet the above conditions for automatic compilation or the function name has been reserved by using the `FORWARD_FUNCTION` statement described below. This restriction is necessary in order to distinguish between function calls and subscripted variable references.

---

For information on how to access routines, see “[Executing Program Files](#)” in Chapter 2 of the *Using IDL* manual.

## Forward Function Definition

Versions of IDL prior to version 5.0 used parentheses to indicate array subscripts. Because function calls use parentheses as well, the IDL compiler is not able to distinguish between arrays and functions by examining the statement syntax.

This problem has been addressed beginning with IDL version 5.0 by the use of square brackets “[ ]” instead of parentheses to specify array subscripts. See “[Array Subscript Syntax: \[ \] vs. \( \)](#)” on page 157 for a discussion of the IDL version 5.0 and later syntax. However, because parentheses are still allowed in array subscripting statements, the need for a mechanism by which the programmer can “reserve” a name for a function that has not yet been defined remains. The `FORWARD_FUNCTION` statement addresses this need.

As mentioned above, ambiguities can arise between function calls and array references when a function has not yet been compiled, or there is no file with the same name as the function found in the IDL path.

For example, attempting to compile the IDL statement:

```
A = xyz(1, COLOR=1)
```

will cause an error if the user-written function XYZ has not been compiled and the filename `xyz.pro` is not found in the IDL path. IDL reports a syntax error, because `xyz` is interpreted as an array variable instead of a function name.

This problem can be eliminated by using the `FORWARD_FUNCTION` statement. This statement has the following syntax:

```
FORWARD_FUNCTION Name1, Name2, ..., NameN
```

where *Name* is the name of a function that has not yet been compiled. Any names declared as forward-defined functions will be interpreted as functions (instead of as variable names) for the duration of the IDL session.

For example, we can resolve the ambiguity in the previous example by adding a `FORWARD_FUNCTION` definition:

```
;Define XYZ as the name of a function that has not yet been
;compiled.
FORWARD_FUNCTION XYZ

;IDL now understands this statement to be a function call instead
;of a bad variable reference.
a = XYZ(1, COLOR=1)
```

---

**Note**

Declaring a function that will be merged into IDL via the `LINKIMAGE` command with the `FORWARD_FUNCTION` statement will not have the desired effect. Routines merged via `LINKIMAGE` are considered by IDL to be built-in routines, and thus need no compilation or declaration. They must, however, be merged with IDL before any routines that call them are compiled.

---

# GOTO Statement

The syntax of the GOTO statement is as follows:

```
GOTO, Label
```

The GOTO statement is used to transfer program control to a point in the program specified by the label. The GOTO statement is generally considered to be a poor programming practice that leads to unwieldy programs. Its use should be avoided. However, for those cases in which the use of a GOTO is appropriate, IDL does provide the GOTO statement.

Note that using a GOTO to jump into the middle of a loop results in an error.

## Warning

---

The user must be careful in programming with GOTO statements. It is not difficult to get into a loop that will never terminate, especially if there is not an escape (or test) within the statements spanned by the GOTO.

---

## Example

An example of the GOTO statement follows:

```
GOTO, JUMP1  
Statements ...  
...  
JUMP1: X = 2000 + Y
```

In the above example, the statement at label JUMP1 is executed after the GOTO statement, skipping any intermediate statements. Note that the label could also occur before the GOTO statement that refers to the label.

GOTO statements are frequently the subjects of IF statements, as seen in the statement below:

```
IF A NE G THEN GOTO, MISTAKE
```

# IF Statement

The syntax of the IF statement is as follows:

```
IF Expression THEN  
IF Expression THEN Statement ELSE Statement
```

The IF statement is used to conditionally execute a statement or a block of statements.

The expression after the “IF” is called the *condition* of the IF statement. This expression (or condition) is evaluated, and if true, the statement following the “THEN” is executed. If the expression evaluates to a *false* value, the statement following the “ELSE” clause is executed. Control passes immediately to the next statement if the condition is false and the ELSE clause is not present.

## Example

Examples of the IF statement include the following:

```
IF A NE 2 THEN PRINT, 'A is not two '  
IF A EQ 1 THEN PRINT, 'A is one' ELSE PRINT, 'not one '
```

The first example contains no ELSE clause. If the value of A is not equal to 2, “A IS NOT TWO” is printed. If A is equal to 2, the THEN clause is ignored, nothing is printed, and execution resumes at the next statement. In the second example above, the condition of the IF statement is (A EQ 1). If the value of A is equal to 1, “A IS ONE” is printed; otherwise, “NOT ONE” is printed.

## Definition of True and False

The condition of the IF statement can be any scalar expression. The definition of *true* and *false* for the different data types is as follows:

- Byte, integer, and long: odd integers are true, even integers are false.
- Floating-Point, double-precision floating-point, and complex: non-zero values are true, zero values are false. The imaginary part of complex numbers is ignored.
- String: any string with a nonzero length is true, null strings are false.

## Example

In the following example, the logical statement for the condition is a conjunction of two conditions.

```
IF (LON GT -40) AND (LON LE -20) THEN ...
```

If both conditions (LON being larger than -40 and less than or equal to -20) are true, the statement following the THEN is executed.

## Using Statement Blocks with the IF Statement

The THEN and ELSE clauses can be in the form of a block (or group of statements) with the delimiters BEGIN and END (see “Blocks” on page 204). To ensure proper nesting of blocks, you can use ENDIF and ENDELSE to terminate the block, instead of using the generic END. Below is an example of the use of blocks within an IF statement.

```
IF (I NE 0.0) THEN BEGIN
    ...
ENDIF ELSE BEGIN
    ...
ENDELSE
```

## Nesting IF Statements

IF statements can be nested in the following manner:

```
IF P1 THEN S1 ELSE $
IF P2 THEN S2 ELSE $
...
IF PN THEN SN ELSE SX
```

If condition P1 is true, only statement S1 is executed; if condition P2 is true, only statement S2 is executed, etc. If none of the conditions are true, statement SX will be executed. Conditions are tested in the order they are written. The construction above is similar to the CASE statement except that the conditions are not necessarily related.

# Procedure Call Statement

The syntax of the procedure call statement is as follows:

```
Procedure_Name, Parameter1, Parameter2, ..., Parametern
```

The procedure call statement invokes a system, user-written, or externally-defined procedure. The parameters that follow the procedure's name are passed to the procedure. When the called procedure finishes, control resumes at the statement following the procedure call statement. Procedure names can be up to 128 characters long.

Procedures can come from the following sources:

- System procedures provided with IDL.
- User-written procedures written in IDL and compiled with the `.RUN` command.
- User-written procedures that are compiled automatically because they reside in directories in the search path. These procedures are compiled the first time they are used. See [“Function Definition Statement”](#) on page 216.
- Procedures written in IDL, that are included with the IDL distribution, located in directories that are specified in the search path.
- Under many operating systems, user-written system procedures coded in FORTRAN, C, or any language that follows the standard calling conventions, which have been dynamically linked with IDL using the `LINKIMAGE` or `CALL_EXTERNAL` procedures.

## Example

Some procedures can be called without any parameters. For example:

```
ERASE
```

This is a procedure call to a subroutine to erase the screen. There are no explicit inputs or outputs. Other procedures have one or more parameters. For example, the statement:

```
PLOT, CIRCLE
```

calls the `PLOT` procedure with the parameter `CIRCLE`.

## Parameter Passing

Parameters passed to procedures and functions are identified by their *position* or by a *keyword*. As their name indicates, the position of positional parameters establishes the correspondence of the parameters in the call and those in the definition of the procedure or function. A keyword parameter is a parameter preceded by a keyword and “=” that identifies the parameter.

Parameters are passed by *value* or by *reference*. Parameters that consist of only a variable name are passed by reference. Expressions, constants, and system variables are passed by value. The two passing mechanisms are fundamentally different. The called procedure or function cannot return a value in a parameter that is passed by value, as the value of the parameter is evaluated and passed into the called procedure, but is not copied back to the caller. Changes made by the called procedure are passed back to the caller if the parameter is passed by reference. See “[Parameter Passing Mechanism](#)” on page 298.

Parameters can be of any type or structure, although some system procedures, as well as user-defined procedures, may require a particular type of parameter for a specific argument. Parameters also can be expressions which are evaluated, used in the call, and then discarded. For example:

```
PLOT, SIN(CIRCLE)
```

The sine of the array CIRCLE is computed and plotted, then the result of the computation is discarded.

### Keyword Parameters

A keyword parameter is a parameter preceded by a keyword and “=” that identifies the parameter.

For example, the PLOT procedure can be manipulated to retain, rather than erase, the image on the screen, as well as to draw, using color index 12. Accomplish these tasks with either of the following calls:

```
PLOT, X, Y, NOERASE = 1, COLOR = 12
```

or

```
PLOT, X, Y, COL = 12, /NOERASE
```

The two calls produce identical code. Keywords can be abbreviated to the shortest nonambiguous string. The construct /KEYWORD is equivalent to setting the keyword parameter to the value 1. For example, /NOERASE is equivalent to NOERASE = 1. In the above examples, the parameter X is the first positional

parameter because it is not preceded by a keyword. The parameter Y is the second positional parameter.

The interpretation of keyword arguments is independent of their order. The placement of keyword arguments does not affect the interpretation of positional parameters—keyword parameters can appear before, after, or in the middle of the positional parameters.

Keyword parameters offer the following advantages over positional parameters:

- Procedures and functions can have a large number of arguments, any of which may be optional. Only those arguments that are actually used need be present in the call.
- It is much easier to remember the names of keyword arguments than to remember the order of positional arguments.
- Additional features can be added to existing procedures and functions without changing the meaning or interpretation of other arguments.



# Procedure Definition Statement

The syntax of the PRO statement is as follows:

```
PRO Procedure_Name, Parameter1, Parameter2, ..., Parametern
```

A sequence of one or more IDL statements can be given a name, compiled, and saved for future use with the procedure definition statement. Once a procedure has been successfully compiled, it can be executed using a procedure call statement interactively from the terminal, from a main program, or from another procedure or function.

The general format for the definition of a procedure is as follows:

```
PRO Name, Parameter1, ..., Parametern
    ;Statements defining procedure.
    Statement1
    Statement2
    ...
;End of procedure definition.
END
```

For more detail on defining procedures see [Chapter 13, “Defining Procedures and Functions”](#).

Calling a user-written procedure that is in a directory in the IDL search path (!PATH) and has the same name as the prefix of the .SAV or .PRO file, causes the procedure to be read from the disk, compiled, and executed without interrupting program execution.

# REPEAT Statement

The syntax of the REPEAT statement is as follows:

```
REPEAT Subject_Statement UNTIL Condition_Expression
```

The REPEAT statement repetitively executes its subject statement until a condition is true. The condition is checked after the subject statement is executed. Therefore, the subject statement is always executed at least once.

## Example

Some examples of how the REPEAT statement can be used are as follows:

```
A = 1
B = 10
REPEAT A = A * 2 UNTIL A GT B
```

This code finds the smallest power of 2 that is greater than B. The subject statement can also be in the form of a block:

```
;Sort array.
REPEAT BEGIN
    ;Set flag to true.
    NOSWAP = 1
    FOR I = 0, N - 2 DO IF arr[I] GT arr[I + 1] THEN BEGIN
        ;Swapped elements, clear flag.
        NOSWAP = 0
        T = arr[I] & arr[I] = arr[I + 1] & arr[I + 1] = T
    ENDIF
;Keep going until nothing is moved.
ENDREP UNTIL NOSWAP
```

The above example sorts the elements of ARR using the inefficient bubble sort method. A more efficient way to sort elements is to use IDL's SORT function.

# WHILE Statement

The syntax of the WHILE statement is as follows:

```
WHILE Expression DO Statement
```

WHILE statements are used to execute a statement repeatedly while a condition remains true. The WHILE statement is similar to the REPEAT statement except that the condition is checked prior to the execution of the statement.

When the WHILE statement is executed, the conditional expression is tested, and if it is true, the statement following the DO is executed. Control then returns to the beginning of the WHILE statement, where the condition is again tested. This process is repeated until the condition is no longer true, at which point the control of the program resumes at the next statement.

In the WHILE statement, the subject is never executed if the condition is initially false.

## Example

An example of a WHILE statement follows:

```
WHILE NOT EOF(1) DO READF, 1, A, B, C
```

In this example, data are read until the end-of-file is encountered.

The next example demonstrates one way to find the first point of an array greater than or equal to a selected value assuming the array is sorted into ascending order.

```
;Initialize index.
I = 0

;Increment X until a point larger than X is found or the end of the
;array is reached
WHILE (arr[I] LT X) AND (I LT N) DO I = I + 1
```

Another way to accomplish the same thing is with the WHERE command, which is used to find the subscripts of the points where ARR[I] is greater than or equal to X.

```
;Subscripts of elements of arr that are greater than or equal to X.
P = WHERE(arr GE X)

;Save first subscript.
I = P(0)
```





# Chapter 11: Pointers

The following topics are covered in this chapter:

---

Overview .....	230	Operations on Pointers .....	239
Heap Variables .....	231	Dangling References .....	243
Creating Heap Variables .....	233	Heap Variable Leakage .....	244
Saving and Restoring Heap Variables .....	234	Pointer Validity .....	246
Pointer Heap Variables .....	235	Freeing Pointers .....	247
IDL Pointers .....	236	Pointer Examples .....	248

# Overview

In order to build linked lists, trees, and other dynamic data structures, it must be possible to access variables via lightweight references that may have more than one name. Further, these names might have different lifetimes, so the lifetime of the variable that actually holds the data must be separate from the lifetime of the tokens that are used to access it.

Beginning with IDL version 5, IDL includes a new *pointer* data type to facilitate the construction of dynamic data structures. Although there are similarities between IDL pointers and machine pointers as implemented in languages such as C, it is important to understand that they are not the same thing. IDL pointers are a high level IDL language concept and do not have a direct one-to-one mapping to physical hardware. Rather than pointing at locations in computer memory, IDL pointers point at *heap variables*, which are special dynamically allocated IDL variables. Heap variables are global in scope, and exist until explicitly destroyed.

## Running the Example Code

The example code used in this chapter is part of the IDL distribution. All of the files mentioned are located in the `doc` subdirectory of the `examples` subdirectory of the main IDL directory. By default, this directory is part of IDL's path; if you have not changed your path, you will be able to run the examples as described here. See [!PATH](#) in the *IDL Reference Guide* for information on IDL's path.

# Heap Variables

Heap variables are a special class of IDL variables that have global scope and explicit user control over their lifetime. They can be basic IDL variables, accessible via pointers, or objects, accessible via object references. (See [Chapter 12, “Object Basics”](#) for more information on IDL objects.) In IDL documentation of pointers and objects, heap variables accessible via pointers are called *pointer heap variables*, and heap variables accessible via object references are called *object heap variables*.

## Note

---

Pointers and object references have many similarities, the strongest of which is that both point at heap variables. It is important to understand that they are not the same type, and cannot be used interchangeably. Pointers and object references are used to solve different sorts of problems. Pointers are useful for building dynamic data structures, and for passing large data around using a lightweight token (the pointer itself) instead of copying data. Objects are used to apply object oriented design techniques and organization to a system. It is, of course, often useful to use both in a given program.

---

Heap variables are global in scope, but do not suffer from the limitations of COMMON blocks. That is, heap variables are available to all program units at all times. (Remember, however, that IDL variables containing pointers to heap variables are *not* global in scope and must be declared in a COMMON block if you want to share them between program units.)

Heap variables:

- Facilitate object oriented programming.
- Provide full support for Save and Restore. Saving a pointer or object reference automatically causes the associated heap variable to be saved as well. This means that if the heap variable contains a pointer or object reference, the heap variables they point to are also saved. Complicated self-referential data structures can be saved and restored easily.
- Are manipulated primarily via pointers or object references using built in language operators rather than special functions and procedures.
- Can be used to construct arbitrary, fully general data structures in conjunction with pointers.

**Note**

---

If you have used versions of IDL prior to version 5, you may be familiar with *handles*. Because IDL pointers provide a more complete and robust way of building dynamic data structures, Research Systems recommends that you use pointers rather than handles when developing new code. See [Appendix H, “Obsolete Routines”](#) in the *IDL Reference Guide* for a discussion of Research Systems’ policy on language features that have been superseded in this manner.

---



## Creating Heap Variables

Heap variables can be created only by the pointer creation function `PTR_NEW` or the object creation function `OBJ_NEW`. (See [Chapter 12, “Object Basics”](#) for a discussion of object creation.) Copying a pointer or object reference *does not* create a new heap variable. This is markedly different from the way IDL handles “regular” variables. For example, with the statement:

```
A = 1.0
```

you create a new IDL floating-point variable with a value of 1.0. The following statement:

```
B = A
```

creates a second variable with the same type and value as A.

In contrast, if you create a new heap variable with the following command:

```
C = PTR_NEW(2.0d)
```

the variable C contains not the double-precision floating-point value 2.0, but a pointer to a heap variable that contains that value. Copying the variable C with the following statement:

```
D = C
```

does not create another heap variable, but rather creates a second pointer to the same heap variable. In this example, the `HELP` command would reveal:

```
% At $MAIN$
A          FLOAT      =      1.00000
B          FLOAT      =      1.00000
C          POINTER    = <PtrHeapVar1>
D          POINTER    = <PtrHeapVar1>
```

The variables C and D are both pointers to the same heap variable. (The actual name assigned to a heap variable is arbitrary.) Changing the value stored in the heap variable would be reflected when dereferencing either C or D (dereferencing is discussed in [“Dereference”](#) on page 239).

Destroying or redefining either C, D, or both variables would leave the contents of the heap variable unchanged. When all pointers or references to a given heap variable are destroyed, the heap variable still exists and holds whatever memory has been allocated for it. See [“Heap Variable Leakage”](#) on page 244 for further discussion. If the heap variable itself is destroyed, pointers to the heap variable may still exist, but will be invalid. See [“Dangling References”](#) on page 243.

## Saving and Restoring Heap Variables

The `SAVE` and `RESTORE` procedures work for heap variables just as they work for all other supported types. When IDL saves a pointer or object reference in a save file, it recursively saves the heap variables that are referenced by that pointer or object reference. `SAVE` handles circular data structures correctly. You can build a large, complicated, self-referential data structure, and then save the entire construct with a call to `SAVE` to save the single pointer or object reference that points to the head of the structure. For example, you can save a pointer to the root of a binary tree and the entire tree will be saved.

The internal identifier of a given heap variable is dynamically allocated at run time, and will differ between IDL sessions. As a result, the `RESTORE` operation maps all saved pointers and object references to their new values in the current session.

# Pointer Heap Variables

*Pointer heap variables* are IDL heap variables that are accessible only via *pointers*. While there are many similarities between object references and pointers, it is important to understand that they are not the same type, and cannot be used interchangeably. Pointer heap variables are created using the [PTR\\_NEW](#) and [PTRARR](#) functions. For more information on objects, see [Chapter 12, “Object Basics”](#).

# IDL Pointers

As illustrated above, you must use a special IDL routine to create a pointer to a heap variable. Two routines are available: `PTR_NEW` and `PTRARR`. Before discussing these functions, however, it is useful to examine the concept of a null pointer.

## Null Pointers

The *Null Pointer* is a special pointer value that is guaranteed to never point at a valid heap variable. It is used by IDL to initialize pointer variables when no other initializing value is present. It is also a convenient value to use at the end nodes in data structures such as trees and linked lists.

It is important to understand the difference between a null pointer and a pointer to an undefined or invalid heap variable. The second case is a valid pointer to a heap variable that does not currently contain a usable value. To make the difference clear, consider the following IDL statements:

```
;The variable A contains a null pointer.
A = PTR_NEW()
;The variable B contains a pointer to a heap variable with an
;undefined value.
B = PTR_NEW(/ALLOCATE_HEAP)

HELP, A, B, *B
```

IDL prints:

```
A          POINTER = <NullPointer>
B          POINTER = <PtrHeapVar1>
<PtrHeapVar1> UNDEFINED = <Undefined>
```

The primary difference is that it is possible to write a useful value into a pointer to an undefined variable, but this is never possible with a null pointer. For example, attempt to assign the value 34 to the null pointer:

```
*A = 34
```

IDL prints:

```
% Unable to dereference NULL pointer: A.
% Execution halted at: $MAIN$
```

Assign the value 34 to a previously-undefined heap variable:

```
*B = 34
PRINT, *B
```

IDL prints:

```
34
```

Similarly, the null pointer is not the same thing as the result of `PTR_NEW(0)`. `PTR_NEW(0)` returns a pointer to a heap variable that has been initialized with the integer value 0.

## The `PTR_NEW` Function

Use the `PTR_NEW` function to create a single pointer to a new heap variable. If you supply an argument, the newly-created heap variable is set to the value of the argument. For example, the command:

```
ptr1 = PTR_NEW(FINDGEN(10))
```

creates a new heap variable that contains the ten-element floating point array created by `FINDGEN`, and places a pointer to this heap variable in `ptr1`.

Note that the argument to `PTR_NEW` can be of any IDL data type, and can include any IDL expression, including calls to `PTR_NEW` itself. For example, the command:

```
ptr2 = PTR_NEW({name:'', next:PTR_NEW()})
```

creates a pointer to a heap variable that contains an anonymous structure with two fields: the first field is a string, the second is a pointer. We will develop this idea further in the examples at the end of this chapter.

If you do not supply an argument, the newly-created pointer will be a null pointer. If you wish to create a new heap variable but do not wish to initialize it, use the `ALLOCATE_HEAP` keyword.

See [PTR\\_NEW](#) in the *IDL Reference Guide* for further details.

## The `PTRARR` Function

Use the `PTRARR` function to create an array of pointers of up to eight dimensions. By default, every element of the array created by `PTRARR` is set to the null pointer. For example:

```
;Create a 2 by 2 array of null pointers.
ptarray = PTRARR(2,2)

;Display the contents of the ptarray variable, and of the first
;array element.
HELP, ptarray, ptarray(0,0)
```

IDL prints:

```
PTARR          POINTER = Array(2, 2)
<Expression>  POINTER = <NullPointer>
```

If you want each element of the array to point to a new heap variable (as opposed to being a null pointer), use the `ALLOCATE_HEAP` keyword. Note that in either case, you will need to initialize the array with another IDL statement.

See [PTRARR](#) in the *IDL Reference Guide* for further details.

# Operations on Pointers

Pointer variables are not directly usable by many of the operators, functions, or procedures provided by IDL. You cannot, for example, do arithmetic on them or plot them. You can, of course, do these things with the heap variables referenced by such pointers, assuming that they contain appropriate data for the task at hand. Pointers exist to allow the construction of dynamic data structures that have lifetimes that are independent of the program scope they are created in.

There are 4 IDL operators that work with pointer variables: assignment, dereference, EQ, and NE. The remaining operators (addition, subtraction, etc.) do not make any sense for pointer types and are not defined.

Many non-computational functions and procedures in IDL do work with pointer variables. Examples are SIZE, N\_ELEMENTS, HELP, and PRINT. It is worth noting that the only I/O allowed directly on pointer variables is default formatted output, where they are printed as a symbolic description of the heap variable they point at. This is merely a debugging aid for the IDL programmer—input/output of pointers does not make sense in general and is not allowed. Please note that this does *not* imply that I/O on the contents of non-pointer data held in heap variables is not allowed. Passing the contents of a heap variable that contains non-pointer data to the PRINT command is a simple example of this type of I/O.

## Assignment

Assignment works in the expected manner—assigning a pointer to a variable gives you another variable with the same pointer. Hence, after executing the statements:

```
A = PTR_NEW(FINDGEN(10))
B = A
HELP, A, B
```

A and B both point at the same heap variable and we see the output:

```
A          POINTER = <PtrHeapVar1>
B          POINTER = <PtrHeapVar1>
```

## Dereference

In order to get at the contents of a heap variable referenced by a pointer variable, you must use the *dereference operator*, which is \* (the asterisk). The dereference operator precedes the variable dereferenced. For example, if you have entered the above assignments of the variables A and B:

```
PRINT, *B
```

IDL prints:

```
0.00000  1.00000  2.00000  3.00000  4.00000  5.00000
6.00000  7.00000  8.00000  9.00000
```

That is, IDL prints the contents of the heap variable pointed at by the pointer variable B.

## Dereferencing Pointer Arrays

Note that the dereference operator requires a *scalar* pointer operand. This means that if you are dealing with a pointer array, you must specify which element to dereference. For example, create a three-element pointer array, allocating a new heap variable for each element:

```
ptarr = PTRARR(3, /ALLOCATE_HEAP)
```

To initialize this array such that the heap variable pointed at by the first pointer contains the integer zero, the second the integer one, and the third the integer two, you would use the following statement:

```
FOR I = 0,2 DO *ptarr[I] = I
```

### Note

The dereference operator is dereferencing only element I of the array for each iteration. Similarly, if you wanted to print the values of the heap variables pointed at by the pointers in ptarr, you might be tempted to try the following:

```
PRINT, *ptarr
```

IDL prints:

```
% Expression must be a scalar in this context: PTARR.
% Execution halted at: $MAIN$
```

To print the contents of the heap variables, use the statement:

```
FOR I = 0, N_ELEMENTS(ptarr)-1 DO PRINT, *ptarr[I]
```

## Dereferencing Pointers to Pointers

The dereference operator can be applied as many times as necessary to access data pointed at indirectly via multiple pointers. For example, the statement:

```
A = PTR_NEW(PTR_NEW(47))
```



assigns to A a pointer to a pointer to a heap variable containing the value 47.

To print this value, use the following statement:

```
PRINT, **A
```

## Dereferencing Pointers within Structures

If you have a structure field that contains a pointer, dereference the pointer by prepending the dereference operator to the front of the structure name. For example, if you define the following structure:

```
struct = {data:'10.0', pointer:ptr_new(20.0)}
```

you would use the following command to print the value of the heap variable pointed at by the pointer in the pointer field:

```
PRINT, *struct.pointer
```

Defining pointers to structures is another common practice. For example, if you define the following pointer:

```
ptstruct = PTR_NEW(struct)
```

you would use the following command to print the value of the heap variable pointed at by the pointer field of the `struct` structure, which is pointed at by `ptstruct`:

```
PRINT, *(*ptstruct).pointer
```

Note that you must dereference both the pointer to the structure and the pointer within the structure.

## Dereferencing the Null Pointer

It is an error to dereference the NULL pointer, an invalid pointer, or a non-pointer. These cases all generate errors that stop IDL execution. For example:

```
PRINT, *45
```

IDL prints:

```
% Pointer type required in this context: <INT(      45)>.
% Execution halted at: $MAIN$
```

For example:

```
A = PTR_NEW() & PRINT, *A
```

IDL prints:

```
% Unable to dereference NULL pointer: A.
% Execution halted at: $MAIN$
```

For example:

```
A = PTR_NEW(23) & PTR_FREE, A & PRINT, *A
```

IDL prints:

```
% Invalid pointer: A.
```

```
% Execution halted at: $MAINS$
```

## Equality and Inequality

The EQ and NE operators allow you to compare pointers to see if they point at the same heap variable. For example:

```
;Make A a pointer to a heap variable containing 23.
A = PTR_NEW(23)

;B points at the same heap variable as A.
B = A

;C contains the null pointer.
C = PTR_NEW()

PRINT, 'A EQ B: ', A EQ B & $
PRINT, 'A NE B: ', A NE B & $
PRINT, 'A EQ C: ', A EQ C & $
PRINT, 'C EQ NULL: ', C EQ PTR_NEW() & $
PRINT, 'C NE NULL: ', C NE PTR_NEW()
```

IDL prints:

```
A EQ B:      1
A NE B:      0
A EQ C:      0
C EQ NULL:   1
C NE NULL:   0
```

# Dangling References

If a heap variable is destroyed, any remaining pointer variable or object reference that still refers to it is said to contain a *dangling reference*. Unlike lower level languages such as C, dereferencing a dangling reference will not crash or corrupt your IDL session. It will, however, fail with an error message. For example:

```
;Create a new heap variable.
A = PTR_NEW(23)

;Print A and the value of the heap variable A points to.
PRINT, A, *A
```

IDL prints:

```
<PtrHeapVar13>      23
```

For example:

```
;Destroy the heap variable.
PTR_FREE, A

;Try to print again.
PRINT, A, *A
```

IDL prints:

```
% Invalid pointer: A.
% Execution halted at: $MAIN$
```

There are several possible approaches to avoiding such errors. The best option is to structure your code such that dangling references do not occur. You can, however, verify the validity of pointers or object references before using them (via the [PTR\\_VALID](#) or [OBJ\\_VALID](#) functions) or use the [CATCH](#) mechanism to recover from the effect of such a dereference.

# Heap Variable Leakage

Heap variables are not reference counted—that is, IDL does not keep track of how many references to a heap variable exist, or stop the last such reference from being destroyed—so it is possible to lose access to them and the memory they are using. For example:

```
;Create a new heap variable.
A = PTR_NEW(23)

;Set the pointer A equal to the integer zero. The pointer to the
;heap variable created with the first command is lost.
A = 0
```

Use the `HEAP_VARIABLES` keyword to the `HELP` procedure to view a list of heap variables currently in memory:

```
HELP, /HEAP_VARIABLES
```

IDL prints:

```
<PtrHeapVar14> INT = 23
```

In this case, the heap variable `<PtrHeapVar14>` exists and has a value of 23, but there is no way to reference the variable. There are two options: manually create a new pointer to the existing heap variable using the `PTR_VALID` function (see [PTR\\_VALID](#) in the *IDL Reference Guide*), or do manual “Garbage Collection” and use the `HEAP_GC` command to destroy all inaccessible heap variables.

## Warning

---

Object reference heap variables are subject to the same problems as pointer heap variables. See [OBJ\\_VALID](#) in the *IDL Reference Guide* for more information.

---

The [HEAP\\_GC](#) procedure causes IDL to hunt for all unreferenced heap variables and destroy them. It is important to understand that this is a potentially computationally expensive operation, and should not be relied on by programmers as a way to avoid writing careful code. Rather, the intent is to provide programmers with a debugging aid when attempting to track down heap variable leakage. In conjunction with the `VERBOSE` keyword, `HEAP_GC` makes it possible to determine when variables have leaked, and it provides some hint as to their origin.

**Warning**

---

HEAP\_GC uses a recursive algorithm to search for unreferenced heap variables. If HEAP\_GC is used to manage certain data structures, such as large linked lists, a potentially large number of operations may be pushed onto the system stack. If so many operations are pushed that the stack runs out of room, IDL will crash.

---

General reference counting, the usual solution to such leaking, is too slow to be provided automatically by IDL, and careful programming can easily avoid this pitfall. Furthermore, implementing a reference counted data structure on top of IDL pointers is easy to do in those cases where it is useful, and such reference counting could take advantage of its domain specific knowledge to do the job much faster than the general case.

Another approach would be to write allocation and freeing routines—layered on top of the `PTR_NEW` and `PTR_FREE` routines—that keep track of all outstanding pointer allocations. Such routines might make use of pointers themselves to keep track of the allocated pointers. Such a facility could offer the ability to allocate pointers in named groups, and might provide a routine that frees all heap variables in a given group. Such an operation would be very efficient, and is easier than reference counting.

# Pointer Validity

Use the `PTR_VALID` function to verify that one or more pointer variables point to valid and currently existing heap variables, or to create an array of pointers to existing heap variables. If supplied with a single pointer as its argument, `PTR_VALID` returns `TRUE` (1) if the pointer argument points at a valid heap variable, or `FALSE` (0) otherwise. If supplied with an array of pointers, `PTR_VALID` returns an array of `TRUE` and `FALSE` values corresponding to the input array. If no argument is specified, `PTR_VALID` returns an array of pointers to all existing pointer heap variables. For example:

```
;Create a new pointer and heap variable.
A = PTR_NEW(10)

IF PTR_VALID(A) THEN PRINT, "A points to a valid heap variable." $
ELSE PRINT, "A does not point to a valid heap variable."
```

IDL prints:

```
A points to a valid heap variable.
```

For example:

```
;Destroy the heap variable.
PTR_FREE, A

IF PTR_VALID(A) THEN PRINT, "A points to a valid heap variable." $
ELSE PRINT, "A does not point to a valid heap variable."
```

IDL prints:

```
A does not point to a valid heap variable.
```

See [PTR\\_VALID](#) in the *IDL Reference Guide* for further details.

# Freeing Pointers

The `PTR_FREE` procedure destroys the heap variables pointed at by pointers supplied as its arguments. Any memory used by the heap variable is released, and the heap variable ceases to exist. `PTR_FREE` is the only way to destroy a pointer heap variable; if `PTR_FREE` is not called on a heap variable, it continues to exist until the IDL session ends, even if no pointers remain to reference it.

Note that the pointers themselves are not destroyed. Pointers that point to nonexistent heap variables are known as dangling references, and are discussed in more detail in [“Dangling References”](#) on page 243.

See `PTR_FREE` in the *IDL Reference Guide* for further details.

# Pointer Examples

Pointers are useful in building dynamic memory structures, such as linked lists and trees. The following examples demonstrate how pointers are used to build several types of dynamic structures. Note that the purpose of these examples is to illustrate simply and clearly how pointers are used. As such, they may not represent the “best” or most efficient way to accomplish a given task. Readers interested in learning more about efficient use of data structures are urged to consult any good text on data structures.

## Creating a Linked List

The following example uses pointers to create and manipulate a linked list. One procedure reads string input from the keyboard and creates a list of pointers to heap variables that have the strings as their values. Another procedure prints the strings, given the pointer to the beginning of the linked list. A third procedure uses a modified “bubble sort” algorithm to reorder the values so the strings are in alphabetical order.

### Creating the List

The following program prompts the user to enter a series of strings from the keyboard. After reading each string, it creates a new heap variable containing a list element—an anonymous structure with two fields; one to hold the string data and one to hold a pointer to the next list element. Any number of strings can be entered. When the user is finished entering strings, the program can be exited by entering a period by itself at the “Enter string:” prompt.

The text of the program shown below can be found in the file `ptr_read.pro` in the `doc` subdirectory of the `examples` subdirectory of the IDL distribution.

```

;PTR_READ accepts one argument, a named variable in which to return
;the pointer that points at the beginning of the list.
PRO ptr_read, first

;Initialize the input string variable.
newstring = ''

;Create an anonymous structure to contain list elements. Note that
;the next field is initialized to be a null pointer.
l1list = {name:'', next:PTR_NEW()}

;Print instructions for this program.
PRINT, 'Enter a list of names.'
PRINT, 'Enter a period (.) to stop list entry.'

```



```

;Continue accepting input until a period is entered.
WHILE newstring NE "." DO BEGIN

    READ, newstring, PROMPT='Enter string: '
;Read a new string from the keyboard.

;Check to see if a pointer called first exists. If not, this is
;the first element. Create a pointer called first and initialize
;it to be a list element. Create a second pointer to the heap
;variable pointed at by first.
    IF newstring NE '.' THEN BEGIN
        IF NOT(PTR_VALID(first)) THEN BEGIN
            first = PTR_NEW(llist)
            current = first
        ENDIF

;Create a pointer to the next list element.
        next = PTR_NEW(llist)

;Set the name field of current to the input string.
        (*current).name = newstring

;Set the next field of current to the pointer to the next list
;element.
        (*current).next = next

;Store the "current" pointer as the "last" pointer.
        last = current

;Make the "next" pointer the "current" pointer.
        current = next

    ENDIF
ENDWHILE

;Set the next field of the last element to the null pointer.
IF PTR_VALID(last) THEN (*last).next = PTR_NEW()

;End of PTR_READ program.
END

```

Run the PTR\_READ program by entering the following command at the IDL prompt:

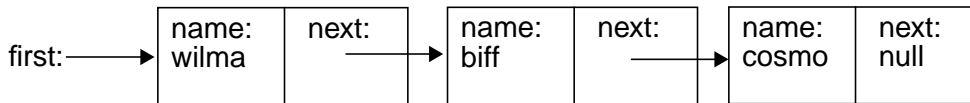
```
ptr_read, first
```

Type a string, press Return, and the program prompts for another string. You can enter as many strings as you want. Each time a string is entered, PTR\_READ creates

a new list element with that string as its value. For example, you could enter the following three strings (used in the rest of this example):

```
Enter a list of names.
Enter a period (.) to stop list entry.
Enter string: wilma
Enter string: biff
Enter string: cosmo
Enter string: .
```

The following figure shows one way of visualizing the linked list that we've created.



*Table 11-1: One way of visualizing the linked list created by the PTR\_READ procedure*

## Printing the Linked List

The next program in our example accepts the pointer to the first element of the linked list and prints all the values in the list in order. To illustrate how the list is linked, we will also print the name of the heap variable that contains each element, and the name of the heap variable in the next field of that element.

The text of the program shown below can be found in the file `ptr_print.pro` in the `doc` subdirectory of the `examples` subdirectory of the IDL distribution.

```

;PTR_PRINT accepts one argument, a pointer to the first element of
;a linked list returned by PTR_READ. Note that the PTR_PRINT
;program does not need to know how many elements are in the list,
;nor does it need to explicitly know of any pointer other than the
;first.
PRO ptr_print, first

;Create a second pointer to the heap variable pointed at by first.
current = first

;PTR_VALID returns 0 if its argument is not a valid pointer. Note
;that the null pointer is not a valid pointer.
WHILE PTR_VALID(current) DO BEGIN

    ;Print the list element information.
    PRINT, current, ', named ', (*current).name, $
  
```

```

    ', has a pointer to: ', (*current).next

    ;Set current equal to the pointer in its own next field.
    current = (*current).next

ENDWHILE

;End of PTR_PRINT program.
END

```

If we run the PTR\_PRINT program with the list generated in the previous example:

```
IDL> ptr_print, first
```

IDL prints:

```

<PtrHeapVar1>, named wilma, has a pointer to: <PtrHeapVar2>
<PtrHeapVar2>, named biff, has a pointer to: <PtrHeapVar3>
<PtrHeapVar3>, named cosmo, has a pointer to: <NullPointer>

```

## A Simple Sorting Routine for the Linked List

The next example program takes a list generated by PTR\_READ and moves the values so that they are in alphabetical order. The sorting algorithm used in this program is a variation on the classic “bubble sort”. However, instead of starting with the last element in the list and letting lower values “rise” to the top, this example starts at the top of the list and lets higher (“heavier”) values “sink” to the bottom of the list. Note that this is not a very efficient sorting algorithm and is shown as an illustration because of its simplicity. For real sorting applications, use IDL’s SORT function.

The text of the program shown below can be found in the file `ptr_sort.pro` in the `doc` subdirectory of the `examples` subdirectory of the IDL distribution.

```

;PTR_SORT accepts one argument, a pointer to the first element of a
;linked list returned by PTR_READ. Note that the PTR_SORT program
;does not need to know how many elements are in the list, nor does
;it need to explicitly know of any pointer other than the first.
pro ptr_sort, first

;Initialize swap flag.
swap = 1

;Create an anonymous structure to contain list elements. Note that
;the next field is initialized to be a pointer.
l1ist = {name:'', next:PTR_NEW()}

;Create a pointer to this structure, to be used as “swap space.”
junk = ptr_new(l1ist)

```

```

;Continue the sorting until no swaps are made. If no adjacent
;elements need to be swapped, the list is in alphabetical order.
WHILE swap NE 0 DO BEGIN

    ;Create a second pointer to the heap variable pointed at by
    ;first.
    current = first

    ;Create another pointer to the heap variable held in the next
    ;field of current.
    next = (*current).next

    ;Set swap flag.
    swap = 0

    ;Continue the sorting until next is no longer a valid pointer.
    ;Note that the null pointer is not a valid pointer.
    WHILE PTR_VALID(next) DO BEGIN

        ;Get values to compare.
        value1 = (*current).name
        value2 = (*next).name

        ;Compare values and exchange if first is greater than second.
        IF (value1 GT value2) THEN BEGIN

            ;Use the "swap space" pointer to exchange the name fields of
            ;current and next.
            (*junk).name = (*current).name
            (*current).name = (*next).name
            (*next).name = (*junk).name

            ;Set current to next to advance through the list.
            current = next

            ;Reset swap flag.
            swap = 1

            ;If value1 is less than value2, set current to next to advance
            ;through the list.
            ENDIF ELSE current = next

            ;Redefine next pointer.
            next = (*current).next
        ENDWHILE
    ENDWHILE
END

```

To run the PTR\_SORT routine with the list from our previous examples as input, enter:

```
ptr_sort, first
```

We can see the results of the sorting by calling the PTR\_PRINT routine again:

```
ptr_print, first
```

IDL prints:

```
<PtrHeapVar1>, named biff, has a pointer to: <PtrHeapVar2>
<PtrHeapVar2>, named cosmo, has a pointer to: <PtrHeapVar3>
<PtrHeapVar3>, named wilma, has a pointer to: <NullPointer>
```

and we see that now the names are in alphabetical order.

## Example Files—Using Pointers to Create Binary Trees

Two more-complicated example programs demonstrate the use of IDL pointers to create and search a simple tree structure. These files, named `idl_tree.pro` and `tree_example.pro`, can be found in the `doc` subdirectory of the `examples` subdirectory of the IDL distribution.

To run the tree examples, enter the following commands at the IDL prompt:

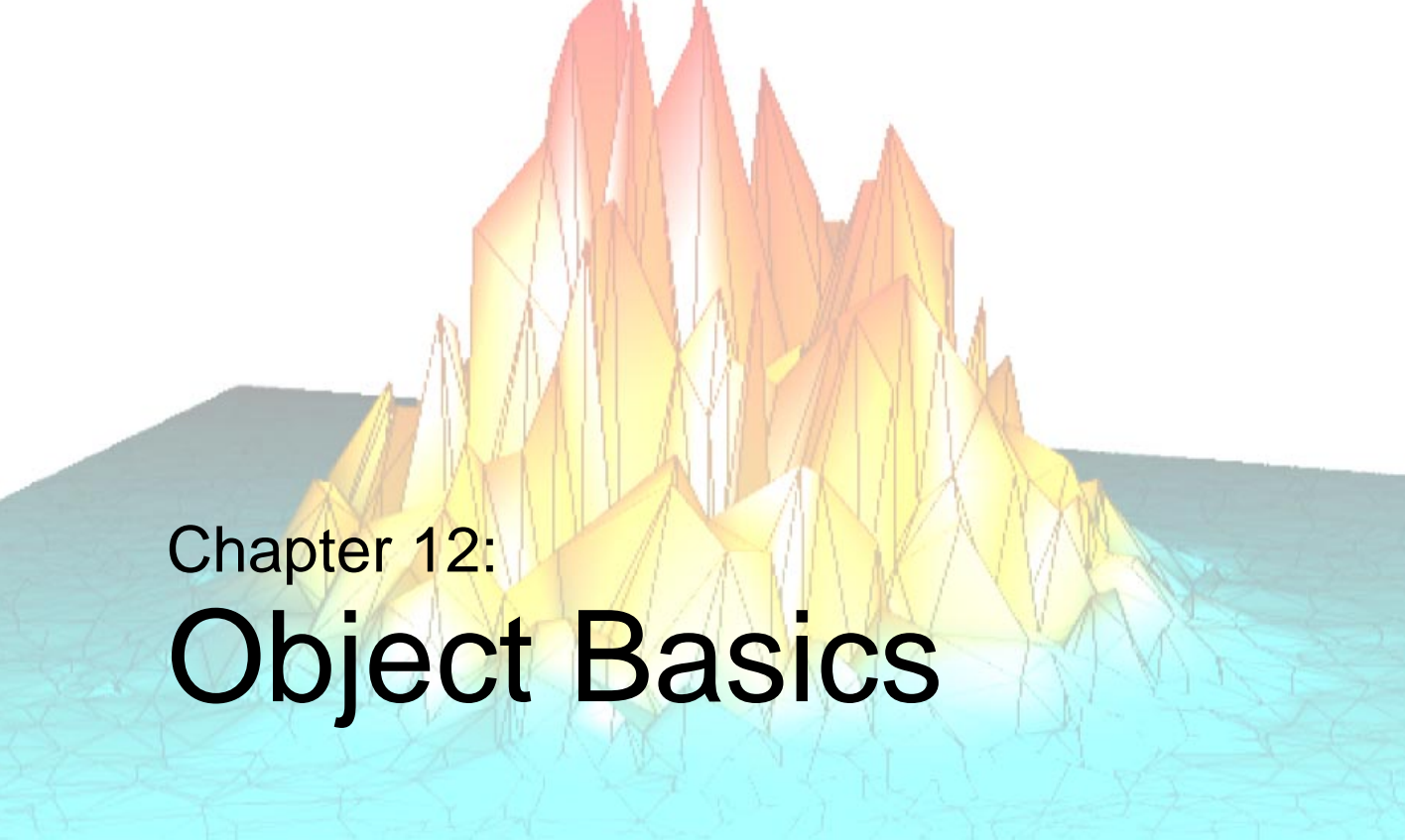
```
;Compile the routines in idl_tree. The example routine calls the
;routines defined in this file.
.run idl_tree

;Run the tree_example.
tree_example
```

The `TREE_EXAMPLE` and `IDL_TREE` routines create a binary tree with ten nodes whose values are structures that contain random values for two fields, “Time” and “Data”. The `TREE_EXAMPLE` routine then prints the tree sorted by both time and data. It then searches for and deletes the nodes containing the fourth and second data values. The resulting 8-node trees are again printed in both time and data order.

A detailed explication of the `TREE_EXAMPLE` and `IDL_TREE` routines is beyond the scope of this chapter. Interested users should examine the files, starting with `tree_example.pro`, to see how the trees are created and searched.





# Chapter 12: Object Basics

The following topics are covered in this chapter:

---

Object-Oriented Programming	256	The Object Lifecycle	266
IDL Object Overview	257	Operations on Objects	269
Class Structures	259	Obtaining Information about Objects	271
Inheritance	261	Method Routines	273
Object Heap Variables	263	Method Overriding	277
Null Objects	265	Object Examples	280

# Object-Oriented Programming

Traditional programming techniques make a strong distinction between routines written in the programming language (procedures and functions in the case of IDL) and data to be acted upon by the routines. *Object oriented* programming begins to remove this distinction by melding the two into *objects* that can contain both routines and data. Object orientation provides a layer of abstraction that allows the programmer to build robust applications from groups of reusable elements.

Beginning in version 5.0, IDL provides a set of tools for developing object-oriented applications. IDL's Object Graphics engine is object-oriented, and a class library of graphics objects allows you to create applications that provide equivalent graphics functionality regardless of your (or your users') computer platform, output devices, etc. As an IDL programmer, you can use IDL's traditional procedures and functions as well as the new object features to create your own object modules. Applications built from object modules are, in general, easier to maintain and extend than their traditional counterparts.

This chapter describes how to use object techniques with IDL. A complete discussion of object orientation is beyond the scope of this book—if you are new to object oriented programming, consult one of the many references on object oriented program that are available.



# IDL Object Overview

IDL objects are actually special *heap variables*, which means that they are global in scope and provide explicit user control over their lifetimes. Object heap variables can only be accessed via object references. Object references are discussed in this chapter. Heap variables in general are discussed in detail in “[Heap Variables](#)” on page 231.

Briefly, IDL provides support for the following object concepts and mechanisms:

## Classes and Instances

IDL objects are created as *instances* of a *class*, which is defined in the form of an IDL structure. The name of the structure is also the class name for the object. The *instance data* of an object is an IDL structure contained in the object heap variable, and can only be accessed by special functions and procedures, called *methods*, which are associated with the class. Class structures are discussed in “[Class Structures](#)” on page 259.

## Encapsulation

*Encapsulation* is the ability to combine data and the routines that affect the data into a single object. IDL accomplishes this by only allowing access to an object’s instance data via that object’s *methods*. Data contained in an object is hidden from all but the object’s own methods.

## Methods

IDL allows you to define method procedures and functions using all of the programming tools available in IDL. Method routines are identified as belonging to an object class via a routine naming convention. Methods are discussed in detail in “[Method Routines](#)” on page 273.

## Polymorphism

*Polymorphism* is the ability to create multiple object types that support the same operations. For example, many of IDL’s graphics objects support an operation called “Draw,” which sends graphics output to a specified place. The “Draw” operation is different in different contexts; sending a graphic to a printer is different from writing it to a file. Polymorphism allows the details of the differences to remain hidden—all you need to know is that a given object supports the “Draw” operation.

## Inheritance

*Inheritance* is the ability of an object class to inherit the behavior of other object classes. This means that when writing a new object class that is very much like an existing object class, you need only program the functions that are different from those in the inherited class. IDL supports multiple inheritance—that is, an object can inherit qualities from any number of other existing object classes. Inheritance is discussed in detail in [“Inheritance”](#) on page 261.

## Persistence

*Persistence* is the ability of objects to remain in existence in memory after they have been created, allowing you to alter their behavior or appearance after their creation. IDL objects persist until you explicitly destroy them, or until the end of the IDL session. In practice, object persistence removes the need (in traditional IDL programs) to re-execute IDL commands that create an item (a plot, for example) in order to change a detail of the item. For example, once you have created a graphic object containing a plot, you can alter any aspect of the plot “on the fly,” without re-creating it. Similarly, having created an object containing a plot, you need not recreate the plot in order to print, save to an image file, or re-display it.

IDL objects also persist in the sense that you can use the `SAVE` and `RESTORE` routines to save and recreate objects between IDL sessions.

# Class Structures

Object instance data is contained in named IDL structures. We will use the term *class structure* to refer to IDL structures containing object instance data.

Beyond the restriction that class structures must be named structures, there are no limits on what a class structure contains. Class structures can include data of any type or organization, including pointers and object references. When an object is created, the name of the class structure becomes the name of the class itself, and thus serves to define the names of all methods associated with the class. For example, if we create the following class structure:

```
struct = { Class1, data1:0L, data2:FLTARR(10) }
```

any objects created from the class structure `Class1` would have the same two fields (`data1`, a long integer, and `data2`, a ten-element floating-point array) and any methods associated with the class would have the name `Class1::method`, where *method* is the actual name of the method routine. Methods are discussed in detail in “[Method Routines](#)” on page 273.

## Note

---

When a new instance of a structure is created from an existing named structure, all of the fields in the newly-created structure are *zeroed*. This means that fields containing numeric values will contain zeros, fields containing string values will contain null strings, and fields containing pointers or objects will contain null pointers or null objects. In other words, no matter what data the original structure contained, the new structure will contain only a template for that type of data. This is true of objects as well; a newly created object will contain a zeroed copy of the class structure as its instance data.

---

It is important to realize that creating a class structure does not create an object. Objects can only be created by calling the `OBJ_NEW` or `OBJARR` function with the name of the class structure as the argument, and can only be accessed via the returned object reference. In addition, object methods can only be called on object, and not on class structures themselves.

## Automatic Class Structure Definition

If IDL finds a reference to a structure that has not been defined, it will search for a structure definition procedure to define it. (This is true of all structure references, not just class structures.) Automatic structure definition is discussed in “[Automatic](#)

[Structure Definition](#)” on page 149. Briefly, if IDL encounters a structure reference for a structure type that has not been defined, it searches for a routine with a name of the form

```
STRUCT__DEFINE
```

where `STRUCT` is the name of the structure type. Note that there are *two* underscores in the name of the structure definition routine.

The following is an example of a structure definition procedure that defines a structure that will be used for the class `CNAME`.

```
PRO CNAME__DEFINE
    struct = { CNAME, data1:0L, data2:FLTARR(10) }
END
```

This defines a structure named `CNAME` with 2 data fields (`data1`, a long integer, and `data2`, a ten-element floating-point array). If you tell IDL to create an object of type `CNAME` before this structure has been defined, IDL will search for the procedure `CNAME__DEFINE` to define the class structure before attempting to create the object. If the `CNAME__DEFINE` procedure has not yet been compiled, IDL will use its normal routine searching algorithm to attempt to find a file named `CNAME__DEFINE.PRO`. If IDL cannot find a defined structure or structure definition routine, the object-creation operation will fail.

---

**Note**

If you are creating structure definitions on the fly, the possibility exists that you will run into namespace conflicts — that is, a structure with the same name as the structure you are attempting to create may already exist. This can be a problem if you are developing object-oriented applications for others, since you probably do not have much control over the IDL environment on your clients’ systems. You can avoid most problems by creating a unique namespace for your routines; Research Systems does this by prefixing the names of objects with the letters “IDL”. To be completely sure that the objects created by your programs are what you expect, however, you should have the program inspect the created structures and handle errors appropriately.

---

# Inheritance

When defining a class structure, use the INHERITS specifier to indicate that this structure inherits instance data and methods from another class structure. For example, if we defined a class structure called “circle,” as follows:

```
struct = { circle, x:0, y:0, radius:0 }
```

we can define a subclass of the “circle” class like this:

```
struct = { filled_circle, color:0, INHERITS circle }
```

You can use the INHERITS specifier in any structure definition. However, when the structure being defined is a *class structure* (that is, an object will be created from the structure), inheritance affects both the structure definition and the object methods available to the object that inherits. The INHERITS specifier is discussed in “[Structure Inheritance](#)” on page 134.

When a class structure inherits from another class structure, it is said to be a *subclass* of the class it inherits from. Similarly, the class that is inherited from is called a *superclass* of the new class. Defining a subclass of an existing class in this manner has two consequences. First, the class structure for the subclass is constructed as if the elements of the inherited class structure were included in-line in the structure definition. In our example, the command defining the “filled\_circle” class above would create the following structure definition:

```
{ filled_circle, color:0, x:0, y:0, radius:0 }
```

Note that the data fields from the inherited structure definition appear in-line at the point where the INHERITS specifier appears.

The second consequence of defining a subclass structure that inherits from another class structure is that when an object is created from the subclass structure, that object inherits the *methods* of the superclass as well as its data fields. That is, if an object of the superclass type has a method, that method is available to objects created from the subclass as well. In our example above, say we create an object of type circle and define a Print method for it. Any objects of type filled\_circle will also have access to the Print method defined for circle.

IDL allows multiple inheritance. This means that you can include the INHERITS specifier as many times as you desire in a structure definition, *as long as all of the resulting data fields have unique names*. Data fields must have unique names because when the class structure definition is built, the tag names are included in-line at the point where the INHERITS specifier appears. Duplicate tag names will cause the

structure definition to fail; it is your responsibility as a programmer to ensure that tag names are not used more than once in a structure definition.

**Note**

---

The requirement that names be unique applies only to *data* fields. It is perfectly legitimate (and often necessary) for subclasses to have methods with the same names as methods belonging to the superclass. See “[Method Overriding](#)” on page 277 for details.

---

If a structure referred to by an INHERITS specifier has not been defined in the current IDL session, IDL will attempt to define it in the manner described in “[Automatic Class Structure Definition](#)” on page 259.

# Object Heap Variables

*Object heap variables* are IDL heap variables that are accessible only via *object references*. While there are many similarities between object references and pointers, it is important to understand that they are not the same type, and cannot be used interchangeably. Object heap variables are created using the OBJ\_NEW and OBJARR functions. For more information on heap variables and pointers, see “[IDL Pointers](#)” on page 236.

Heap variables are a special class of IDL variables that have global scope and explicit user control over their lifetime. They can be basic IDL variables, accessible via pointers, or objects, accessible via object references. In IDL documentation of pointers and objects, heap variables accessible via pointers are called *pointer heap variables*, and heap variables accessible via object references are called *object heap variables*.

## Note

---

Pointers and object references have many similarities, the strongest of which is that both point at heap variables. It is important to understand that they are not the same type, and cannot be used interchangeably. Pointers and object references are used to solve different sorts of problems. Pointers are useful for building dynamic data structures, and for passing large data around using a lightweight token (the pointer itself) instead of copying data. Objects are used to apply object oriented design techniques and organization to a system. It is, of course, often useful to use both in a given program.

---

Heap variables are global in scope, but do not suffer from the limitations of COMMON blocks. That is, heap variables are available to all program units at all times. (Remember, however, that IDL variables containing pointers to heap variables are *not* global in scope and must be declared in a COMMON block if you want to share them between program units.)

Heap variables:

- Facilitate object oriented programming.
- Provide full support for Save and Restore. Saving a pointer or object reference automatically causes the associated heap variable to be saved as well. This means that if the heap variable contains a pointer or object reference, the heap variables they point to are also saved. Complicated self-referential data structures can be saved and restored easily.

- Are manipulated primarily via pointers or object references using built in language operators rather than special functions and procedures.
- Can be used to construct arbitrary, fully general data structures in conjunction with pointers.

## Dangling References

If a heap variable is destroyed, any remaining pointer variable or object reference that still refers to it is said to contain a *dangling reference*. Unlike lower level languages such as C, dereferencing a dangling reference will not crash or corrupt your IDL session. It will, however, fail with an error message.

There are several possible approaches to avoiding such errors. The best option is to structure your code such that dangling references do not occur. You can, however, verify the validity of pointers or object references before using them (via the [PTR\\_VALID](#) or [OBJ\\_VALID](#) functions) or use the [CATCH](#) mechanism to recover from the effect of such a dereferencing.

## Heap Variable “Leakage”

Heap variables are not reference counted—that is, IDL does not keep track of how many references to a heap variable exist, or stop the last such reference from being destroyed—so it is possible to lose access to them and the memory they are using. See [“Heap Variables”](#) on page 231 for additional details.



# Null Objects

The *Null Object* is a special object reference that is guaranteed to never point at a valid object heap variable. It is used by IDL to initialize object reference variables when no other initializing value is present. It is also a convenient value to use when defining structure definitions for fields that are object references, since it avoids the need to have a pre-existing valid object reference.

Null objects are created when you call an object-creation routine but do not specify a class structure to be used as the new object's template. The following statement creates a null object:

```
nullobj = OBJ_NEW()
```

# The Object Lifecycle

As discussed above, objects are *persistent*, meaning they exist in memory until you destroy them. We can break the life of an object into three phases: creation and initialization, use, and destruction. Object *lifecycle routines* allow the creation and destruction of object references; *lifecycle methods* associated with an object allow you to control what happens when an object is created or destroyed.

This section will discuss the first and last phases of the object lifecycle; the remainder of this chapter discusses manipulation of existing objects and use of object method routines.

## Creation and Initialization

Object references are created using one of two lifecycle routines: OBJ\_NEW or OBJARR. Newly created objects are initialized upon creation in two ways:

1. The object reference is created based on the class structure specified,
2. The object's INIT method (if it has one) is called to initialize the object's instance data (contained in fields defined by the class structure). If the object does not have an INIT method, the object's superclasses (if any) are searched for an INIT method.

### The INIT Method

An object's lifecycle method INIT is a function named *Class::INIT* (where *Class* is the actual name of the class). The purpose of the INIT method is to populate a newly-created object with instance data. INIT should return a scalar TRUE value (such as 1) if the initialization is successful, and FALSE (such as 0) if the initialization fails.

The INIT method is unusual in that it *cannot be called outside an object-creation operation*. This means that—unlike most object methods—you cannot call the INIT method on an object directly. You can, however, call an object's INIT method from within the INIT method of a subclass of that object. This allows you to specify parameters used by the superclass' INIT method along with those used by the INIT method of the object being created. In practice, this is often done using the `_EXTRA` keyword. See “[Keyword Inheritance](#)” on page 291 for details.

### The OBJ\_NEW Function

Use the OBJ\_NEW function to create an object reference to a new object heap variable. If you supply the name of a class structure as its argument, OBJ\_NEW creates a new object containing an instance of that class structure. Note that the fields

of the newly-created object's instance data structure will all be empty. For example, the command:

```
obj1 = OBJ_NEW('ClassName')
```

creates a new object heap variable that contains an instance of the class structure *ClassName*, and places an object reference to this heap variable in `obj1`. If you do not supply an argument, the newly-created object will be a null object.

When creating an object from a class structure, `OBJ_NEW` goes through the following steps:

1. If the class structure has not been defined, IDL will attempt to find and call a procedure to define it automatically. See [“Automatic Class Structure Definition”](#) on page 259 for details. If the structure is still not defined, `OBJ_NEW` fails and issues an error.
2. If the class structure has been defined, `OBJ_NEW` creates an object heap variable containing a zeroed instance of the class structure.
3. Once the new object heap variable has been created, `OBJ_NEW` looks for a *method* function named *Class::INIT* (where *Class* is the actual name of the class). If an `INIT` method exists, it is called with the new object as its implicit `SELF` argument, as well as any arguments and keywords specified in the call to `OBJ_NEW`. If the class has no `INIT` method, the usual method-searching rules are applied to find one from a superclass. For more information on methods and method-searching rules, see [“Method Routines”](#) on page 273.

---

**Note**

`OBJ_NEW` does not call all the `INIT` methods in an object's class hierarchy. Instead, it simply calls the first one it finds. Therefore, the `INIT` method for a class should call the `INIT` methods of its direct superclasses as necessary.

---

4. If the `INIT` method returns true, or if no `INIT` method exists, `OBJ_NEW` returns an object reference to the heap variable. If `INIT` returns false, `OBJ_NEW` destroys the new object and returns the `NULL` object reference, indicating that the operation failed. Note that in this case the `CLEANUP` method is not called.

See [`OBJ\_NEW`](#) in the *IDL Reference Guide* for further details.

## The OBJARR Function

Use the `OBJARR` function to create an array of objects of up to eight dimensions. Every element of the array created by `OBJARR` is set to the null object. For example,

the following command creates a 3 by 3 element object reference array with each element contain the null object reference:

```
obj2 = OBJARR(3, 3)
```

See [OBJARR](#) in the *IDL Reference Guide* for further details.

## Destruction

Use the `OBJ_DESTROY` procedure to destroy an object. If the object's class, or one of its superclasses, supplies a procedure method named `CLEANUP`, that method is called, and all arguments and keywords passed by the user are passed to it. The `CLEANUP` method should perform any required cleanup on the object and return. Whether a `CLEANUP` method actually exists or not, IDL will destroy the heap variable representing the object and return.

The `CLEANUP` method is unusual in that it *cannot be called outside an object-destruction operation*. This means that—unlike most object methods—you cannot call the `CLEANUP` method on an object directly. You can, however, call an object's `CLEANUP` method from within the `CLEANUP` method of a subclass of that object.

Note that the object references themselves are not destroyed. Object references that refer to nonexistent object heap variables are known as dangling references, and are discussed in more detail in [“Dangling References”](#) on page 243.

See [OBJ\\_DESTROY](#) in the *IDL Reference Guide* for further details.

# Operations on Objects

Object reference variables are not directly usable by many of the operators, functions, or procedures provided by IDL. You cannot, for example, do arithmetic on them or plot them. You can, of course, do these things with the contents of the structures contained in the object heap variables referred to by object references, assuming that they contain non-object data.

There are four IDL operators that work with object reference variables: assignment, method invocation, EQ, and NE. In addition, the structure dot operator (.) is allowed *within methods of a class*. The remaining operators (addition, subtraction, etc.) do not make any sense for object references and are not defined.

Many non-computational functions and procedures in IDL do work with object references. Examples are SIZE, N\_ELEMENTS, HELP, and PRINT. It is worth noting that the only I/O allowed directly on object reference variables is default formatted output, in which they are printed as a symbolic description of the heap variable they refer to. This is merely a debugging aid for the IDL programmer—input/output of object reference variables does not make sense in general and is not allowed. Please note that this does *not* imply that I/O on the contents of non-object instance data contained in heap variables is not allowed. Passing non-object instance data contained in an object heap variable to the PRINT command is a simple example of this type of I/O.

## Assignment

Assignment works in the expected manner—assigning an object reference to a variable gives you another variable with the same reference. Hence, after executing the statements:

```

;Define a class structure.
struct = { cname, data1:0.0 }

;Create an object.
A = OBJ_NEW('cname')

;Create a second object reference.
B = A

HELP, A, B

```

IDL prints:

```

A          OBJREF    = <ObjHeapVar1(CNAME)>
B          OBJREF    = <ObjHeapVar1(CNAME)>

```

Note that both A and B are references to the same object heap variable.

## Method Invocation

In order to perform an action on an object's instance data, you must call one of the object's *methods*. (See “[Method Routines](#)” on page 273 for more on methods.) To call a method, you must use the method invocation operator, `->` (the hyphen followed by the greater-than sign). The syntax is:

*ObjRef -> Method*

where *ObjRef* is an object reference and *Method* is a method belonging either to the object's class or to one of its superclasses. *Method* may be specified either partially (using only the method name) or completely using both the class name and method name, connected with two colons:

*ObjRef -> Class::Method*

## Equality and Inequality

The EQ and NE operators allow you to compare object references to see if they refer to the same object heap variable. For example:

```
;Define a class structure.
struct = {cname, data:0.0}

;Create an object.
A = OBJ_NEW('CNAME')

;B refers to the same object as A.
B = A

;C contains a null object reference.
C = OBJ_NEW()

PRINT, 'A EQ B: ', A EQ B & $
PRINT, 'A NE B: ', A NE B & $
PRINT, 'A EQ C: ', A EQ C & $
PRINT, 'C EQ NULL: ', C EQ OBJ_NEW() & $
PRINT, 'C NE NULL: ', C NE OBJ_NEW()
```

IDL prints:

```
A EQ B:      1
A NE B:      0
A EQ C:      0
C EQ NULL:   1
C NE NULL:   0
```

# Obtaining Information about Objects

Three IDL routines allow you to obtain information about an existing object:

## OBJ\_CLASS

Use the OBJ\_CLASS function to obtain the class name of a specified object, or to obtain the names of a specified object's direct superclasses. For example, if we create the following class structures:

```
struct = {class1, data1:0.0 }  
struct = {class2, data2a:0, data2b:0L, INHERITS class1 }
```

We can now create an object and use OBJ\_CLASS to determine its class and superclass membership.

```
;Create an object.  
A = OBJ_NEW('class2')  
  
;Print A's class membership.  
PRINT, OBJ_CLASS(A)
```

IDL prints:

```
CLASS2
```

Or you can print as superclasses:

```
;Print A's superclasses.  
PRINT, OBJ_CLASS(A, /SUPERCLASS)
```

IDL prints:

```
CLASS1
```

See [OBJ\\_CLASS](#) in the *IDL Reference Guide* for further details.

## OBJ\_ISA

Use the OBJ\_ISA function to determine whether a specified object is an instance or subclass of a specified object. For example, if we have defined the object A as above:

```
IF OBJ_ISA(A, 'class2') THEN $  
    PRINT, 'A is an instance of class2.'
```

IDL prints:

```
A is an instance of class2.
```

See [OBJ\\_ISA](#) in the *IDL Reference Guide* for further details.

## OBJ\_VALID

Use the `OBJ_VALID` function to verify that one or more object references refer to valid and currently existing object heap variables. If supplied with a single object reference as its argument, `OBJ_VALID` returns `TRUE` (1) if the reference refers to a valid object heap variable, or `FALSE` (0) otherwise. If supplied with an array of object references, `OBJ_VALID` returns an array of `TRUE` and `FALSE` values corresponding to the input array. For example:

```
;Create a class structure.
struct = {cname, data:0.0}

;Create a new object.
A = OBJ_NEW('CNAME')

IF OBJ_VALID(A) PRINT, "A refers to a valid object." $
    ELSE PRINT, "A does not refer to a valid object."
```

IDL prints:

```
A refers to a valid object.
```

If we destroy the object:

```
;Destroy the object.
OBJ_DESTROY, A

IF OBJ_VALID(A) PRINT, "A refers to a valid object." $
    ELSE PRINT, "A does not refer to a valid object."
```

IDL prints:

```
A does not refer to a valid object.
```

See [OBJ\\_VALID](#) in the *IDL Reference Guide* for further details.



# Method Routines

IDL objects can have associated procedures and functions called *methods*. Methods are called on objects via their object references using the method invocation operator.

While object methods are constructed in the same way as any other IDL procedure or function, they are different from other routines in the following ways:

- Object methods are defined using a special naming convention that incorporates the name of the class to which the method belongs.
- All method routines automatically pass an implicit argument named `self`, which contains the object reference of the object on which the method is called.
- Object methods cannot be called on their own. You must use the method invocation operator and supply a valid object reference, either of the class the method belongs to or of one of that class' subclasses.

## Defining Method Routines

Method routines are defined in the same way as other IDL procedures and functions, with the exception that the name of the class to which they belong, along with two colons, is prepended to the method name:

```
PRO ClassName::Method
    IDL statements
END
```

or

```
FUNCTION ClassName::Method, Argument1
    IDL statements
RETURN, value
END
```

For example, suppose we create two objects, each with its own “print” method.

First, define two class structures:

```
struct = { class1, data1:0.0 }
struct = { class2, data2a:0, data2b:0L, INHERITS class1 }
```

Now we define two “print” methods to print the contents of any objects of either of these two classes. (If you are typing this at the IDL command line, enter the `.RUN` command before each of the following procedure definitions.)

```
PRO class1::Print1
```

```

        PRINT, self.data1
    END
    PRO class2::Print2
        PRINT, self.data1
        PRINT, self.data2a, self.data2b
    END

```

Once these procedures are defined, any objects of class1 have access to the method Print1, and any objects of class2 have access to both Print1 and Print2 (because class2 is a subclass of—it *inherits* from—class1). Note that the Print2 method prints the data1 field inherited from class1.

### Note

It is not necessary to give different method names to methods from different classes, as we have done here with Print1 and Print2. In fact, in most cases both methods would have simply been called Print, with each object class knowing only about its own version of the method. We have given the two procedures different names here for instructional reasons; see “[Method Overriding](#)” on page 277 for a more complete discussion of method naming.

## The Implicit *Self* Argument

Every method routine has an implicit argument parameter named *self*. The *self* parameter always contains the object reference of the object on which the method is called. In the method routines created above, *self* is used to specify which object the data fields should be printed from.

You do not need to explicitly pass the *self* argument; in fact, if you try to specify an argument called *self* when defining a method routine, IDL will issue an error.

## Calling Method Routines

You must use the method invocation operator (*->*) to call a method on an object. The syntax is slightly different from other routine invocations:

```

;For a procedure method.
ObjRef -> Method

;For a function method.
Result = ObjRef -> Method()

```

Where *ObjRef* is an object reference belonging to the same class as the *Method*, or to one of that class’ subclasses. We can illustrate this behavior using the Print1 and Print2 methods defined above.

First, define two new objects:

```
A = OBJ_NEW('class1')
B = OBJ_NEW('class2')
```

We can call Print1 on the object A as follows:

```
A -> Print1
```

IDL prints:

```
0.00000
```

Similarly, we can call Print2 on the object B:

```
B -> Print2
```

IDL prints:

```
0.00000
0          0
```

Since the object B inherits its properties from class1, we can also call Print1 on the object B:

```
B -> Print1
```

IDL prints:

```
0.00000
```

We cannot, however, call Print2 on the object A, since class1 does not inherit the properties of class2:

```
A -> Print2
```

IDL prints:

```
% Attempt to call undefined method: 'CLASS1::PRINT2'.
```

## Searching for Method Routines

When a method is called on an object reference, IDL searches for it as with any procedure or function, and calls it if it can be found, following the naming convention established for structure definition routines. (See [“Automatic Class Structure Definition”](#) on page 259.) In other words, IDL discovers methods as it needs them in the same way as regular procedures and functions, with the exception that it searches for files named

```
classname__method.pro
```

rather than simply

`method.pro`

Remember that there are two *underscores* in the file name, and two *colons* in the method routine's name. See [“Executing Program Files”](#) in Chapter 2 of *Using IDL* for details.

---

**Note**

If you are working in an environment where the length of filenames is limited, you may want to consider defining all object methods in the same `.pro` file you use to define the class structure. This practice avoids any problems caused by the need to prepend the *classname* and the two underscore characters to the method name. If you must use different `.pro` files, make sure that all class (and superclass) definition filenames are unique in the first eight characters.

---

# Method Overriding

Unlike data fields, method names can be duplicated. This is an important feature that allows method overriding, which in turn facilitates polymorphism in the design of object-oriented programs. Method overriding allows a subclass to provide its own implementation of a method already provided by one of its superclasses. When a method is called on an object, IDL searches for a method of that class with that name. If found, the method is called. If not, the methods of any inherited object classes are examined in the order their INHERITS specifiers appear in the structure definition, and the first method found with the correct name is called. If no method of the specified name is found, an error occurs.

The method search proceeds *depth first, left to right*. This means that if an object's class does not provide the method called directly, IDL searches through inherited classes by first searching the leftmost included class—and all of its superclasses—before proceeding to the next inherited class to the right. If a method is defined by more than a single inherited structure definition, the first one found is used and no warning is generated. This means that class designers should pick non-generic names for their methods as well as their data fields. For example, suppose we have defined the following classes:

```
struct = { class1, data1 }
struct = { class2, data2a:0, data2b:0.0, inherits class1 }
struct = { class3, data3:'', inherits class2, inherits class1 }
struct = { class 4, data4:0L, inherits class2, inherits class3 }
```

Furthermore, suppose that both `class1` and `class3` have a method called `Print` defined.

Now suppose that we create an object of `class4`, and call the `Print` method:

```
A = OBJ_NEW('class4')
A -> Print
```

IDL takes the following steps:

1. Searches `class4` for a `Print` method. It does not find one.
2. Searches the leftmost inherited class (`class2`) in the class definition structure for a `Print` method. It does not find one.
3. Searches any superclasses of `class2` for a `Print` method. It finds the `class1` `Print` method and calls it on `A`.

Notice that IDL stops searching when it finds a method with the proper name. Thus, IDL doesn't find the `Print` method that belongs to `class3`.

## Specifying Class Names in Method Calls

If you specify a class name when calling an object method, like so:

```
ObjRef -> classname::method
```

Where *classname* is the name of one of the object's superclasses, IDL will search *classname* and any of *classname*'s superclasses for the method name. IDL will *not* search the object's own class or any other classes the object inherits from.

This type of method call is especially useful when a class has a method that overrides a superclass method and does its job by calling the superclass method and then adding functionality. In our simple example from [“Calling Method Routines”](#) on page 274, above, we could have defined a `Print` method for each class, as follows:

```
PRO class1::Print
    PRINT, self.data1
END
PRO class2::Print
    self -> class1::Print
    PRINT, self.data2a, self.data2b
END
```

In this case, to duplicate the behavior of the `Print1` and `Print2` methods, we make the following method calls:

```
A -> Print
```

IDL prints:

```
0.00000
```

And now the B:

```
B -> Print
```

IDL prints:

```
0.00000
0          0
```

Now we'll use the second method:

```
B -> class1::Print
```

IDL prints:

```
0.00000
```

And now A:

```
A -> class2::Print
```

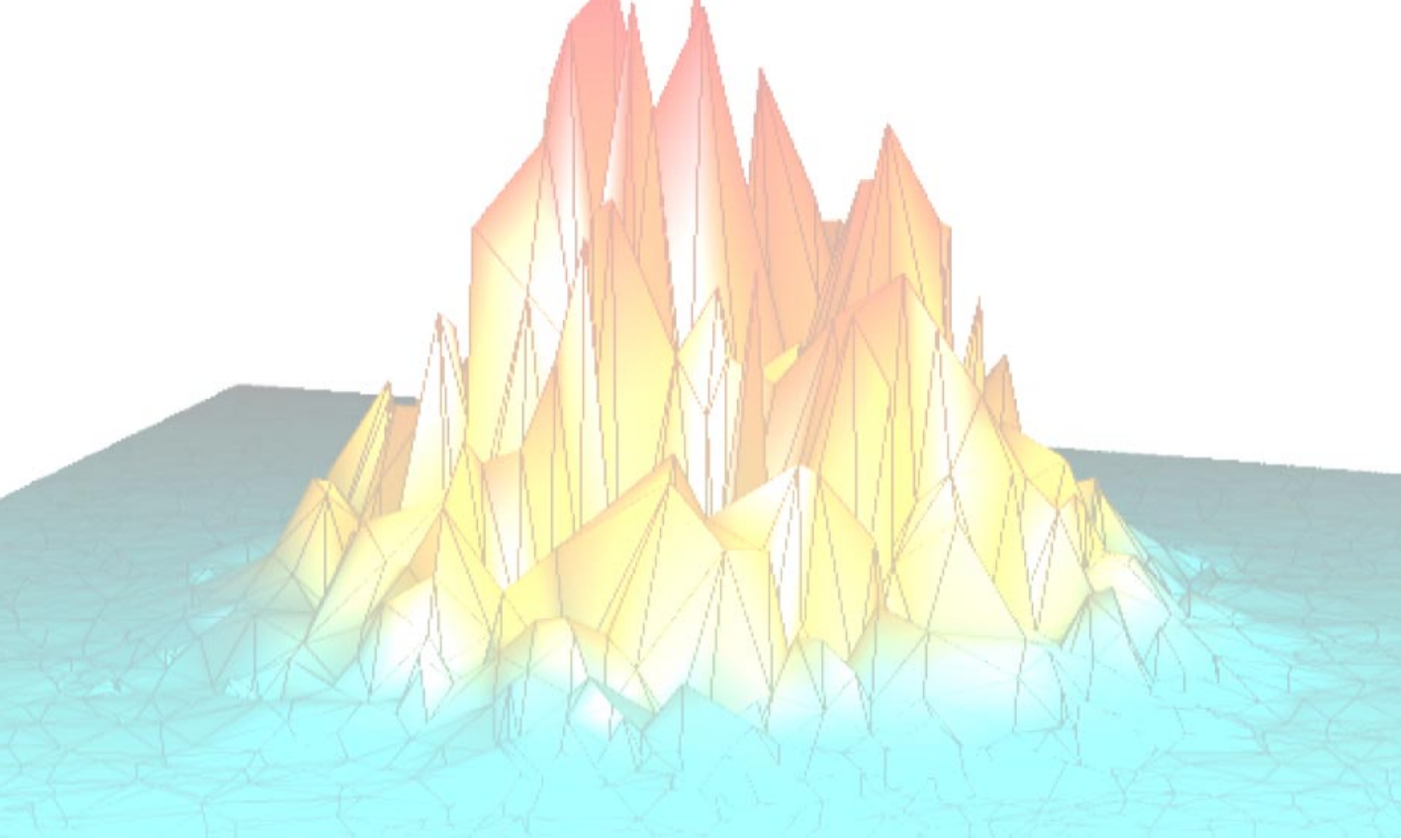
IDL prints:

```
% CLASS2 is not a superclass of object class CLASS1.  
% Execution halted at: $MAIN$
```

## Object Examples

We have included a number of examples of object-oriented programming as part of the IDL distribution. Many of the examples used in this volume are included—sometimes in expanded form—in the object subdirectory of the examples subdirectory of the main IDL directory. By default, this directory is part of IDL's path; if you have not changed your path, you will be able to run the examples as described here. See [!PATH](#) in the *IDL Reference Guide* for information on IDL's path.





# ***Part III: Programming in IDL***





# Chapter 13: Defining Procedures and Functions

The following topics are covered in this chapter:

---

Overview .....	284	Keyword Inheritance .....	291
Procedure & Function Definitions .....	285	Entering Procedure Definitions .....	296
Parameters .....	286	Parameter Passing Mechanism .....	298
Using Keyword Parameters .....	289	Calling Mechanism .....	300

## Overview

Procedures and functions are self-contained modules that break large tasks into smaller, more manageable ones. Modular programs simplify debugging and maintenance and, because they are reusable, minimize the amount of new code required for each application.

New procedures and functions can be written in IDL and called in the same manner as the system-defined procedures or functions from the keyboard or from other programs. When a procedure or function is finished, it executes a `RETURN` statement that returns control to its caller. Functions always return an explicit result. A procedure is called by a procedure call statement, while a function is called by a function reference. For example, if `ABC` is a procedure and `XYZ` is a function, the calling syntax is:

```
;Call procedure ABC with two parameters.  
ABC, A, 12
```

```
;Call function XYZ with one parameter. The result of XYZ is stored  
;in variable A.  
A = XYZ(C/D)
```

## Procedure & Function Definitions

The procedure and function definition statements notify the IDL compiler that a user-written program module follows. One of these statements must be the first line in a user-written IDL routine.

The PRO statement defines an IDL procedure. Its syntax is:

```
PRO Procedure_Name, P1, P2, ..., Pn
```

The FUNCTION statement defines an IDL function. It's syntax is:

```
FUNCTION Function_Name, P1, P2, ..., Pn
```

# Parameters

The variables and expressions passed to the function or procedure from its caller are *parameters*. *Actual parameters* are those appearing in the procedure call statement or the function reference. In the examples at the beginning of this section, the actual parameters in the procedure call are the variable A and the constant 12, while the actual parameter in the function call is the value of the expression (C/D).

*Formal parameters* are the variables declared in the procedure or function definition. The same procedure or function can be called using different actual parameters from a number of places in other program units.

## Correspondence of Formal and Actual Parameters

The correspondence between the actual parameters of the caller and the formal parameters of the called procedure is established by position or by keyword.

### Positional Parameters

A positional parameter, or plain *argument*, is a parameter without a keyword. Just as its name implies, the position of a positional parameter establishes the correspondence—the *n*-th formal positional parameter is matched with the *n*-th actual positional parameter.

### Keyword Parameters

A keyword parameter, which can be either actual or formal, is an expression or variable name preceded by a keyword and an equal sign (“=”) that identifies which parameter is being passed.

When calling a routine with a keyword parameter, you can abbreviate the keyword to its shortest, unambiguous abbreviation. Keyword parameters can also be specified by the caller with the syntax /KEYWORD, which is equivalent to setting the keyword parameter to 1 (e.g., KEYWORD = 1). The syntax /KEYWORD is often referred to, in the rest of this documentation, as *setting* the keyword.

For example, a procedure is defined with a keyword parameter named TEST.

```
PRO XYZ, A, B, TEST = T
```

The caller can supply a value for the formal (keyword) parameter T with the following calls:

```
;Supply only the value of T. A and B are undefined inside the
;procedure.
XYZ, TEST = A
```

```

;The value of A is copied to formal parameter T (note the
;abbreviation for TEST), Q to A, and R to B.
XYZ, TE = A, Q, R

;Variable Q is copied to formal parameter A. B and T are undefined
;inside the procedure.
XYZ, Q

```

---

**Note**

When supplying keyword parameters for a function, the keyword is specified *inside* the parentheses:

---

```
result = FUNCTION(Arg1, Arg2, KEYWORD = value)
```

## Copying Parameters

When a procedure or function is called, the actual parameters are copied into the formal parameters of the procedure or function and the module is executed.

On exit, via a RETURN statement, the formal parameters are copied back to the actual parameters, providing they were not expressions or constants. Parameters can be inputs to the program unit; they can be outputs in which the values are set or changed by the program unit; or they can be both inputs and outputs.

When a RETURN statement is encountered in the execution of a procedure or function, control is passed back to the caller immediately after the point of the call. In functions, the parameter of the RETURN statement is the result of the function.

## Number of Parameters

A procedure or a function can be called with fewer arguments than were defined in the procedure or function. For example, if a procedure is defined with 10 parameters, the user or another procedure can call the procedure with 0 to 10 parameters.

Parameters that are not used in the actual argument list are set to be undefined upon entering the procedure or function. If values are stored by the called procedure into parameters not present in the calling statement, these values are discarded when the program unit exits. The number of actual parameters in the calling list can be found by using the system function `N_PARAMS`. Use the `N_ELEMENTS` function to determine if a variable is defined.

## Example

An example of an IDL function to compute the digital gradient of an image is shown in the example below. The digital gradient approximates the two-dimensional gradient of an image and emphasizes the edges.

This simple function consists of three lines corresponding to the three required components of IDL procedures and functions: the procedure or function declaration, the body of the procedure or function, and the terminating end statement.

```
FUNCTION GRAD, image
;Define a function called GRAD. Result is ABS(dz/dx) + ABS(dz/dy).

;Evaluate and return the result.
  RETURN, ABS(image - SHIFT(image, 1, 0)) + $
          ABS(image-SHIFT(image, 0, 1))

;End of function.
END
```

The function has one parameter called **IMAGE**. There are no local variables. Local variables are variables active only within a module (i.e., they are not parameters and are not contained in common blocks).

The result of the function is the value of the expression used as an argument to the **RETURN** statement. Once compiled, the function is called by referring to it in an expression. Two examples are shown below.

```
;Store gradient of B in A.
A = GRAD(B)

;Display gradient of IMAGE sum.
TVSCL, GRAD(abc + def)
```



## Using Keyword Parameters

A short example of a function that exchanges two columns of a  $4 \times 4$  homogeneous, coordinate-transformation matrix is shown below. The function has one positional parameter, the coordinate-transformation matrix *T*. The caller can specify one of the keywords *XYEXCH*, *XZEXCH*, or *YZEXCH* to interchange the *xy*, *xz*, or *yz* axes of the matrix. The result of the function is the new coordinate transformation matrix defined below.

```

;Function to swap columns of T. XYEXCH swaps columns 0 and 1,
;XZEXCH swaps 0 and 2, and YZEXCH swaps 1 and 2.
FUNCTION SWAP, T, XYEXCH = xy, XZEXCH = xz, YZEXCH = yz

    ;Swap columns 0 and 1 if keyword XYEXCH is set.
    IF KEYWORD_SET(XY) THEN S=[0,1] $

    ;Check to see if xz is set.
    ELSE IF KEYWORD_SET(XZ) THEN S=[0,2] $

    ;Check to see if yz is set.
    ELSE IF KEYWORD_SET(YZ) THEN S=[1,2] $

    ;If nothing is set, return.
    ELSE RETURN, T

    ;Copy matrix for result.
    R = T

    ;Exchange two columns using matrix insertion operators and
    ;subscript ranges.
    R[S[1], 0] = T[S[0], *]
    R[S[0], 0] = T[S[1], *]

    ;Return result.
    RETURN, R

END

```

Typical calls to *SWAP* are as follows:

```

Q = SWAP(!P.T, /XYEXCH)
Q = SWAP(Q, /XYEX)
Q = SWAP(INVERT(Z), YZ = 1)
Q = SWAP(Z, XYE = I EQ 0, XZE = I EQ 1, YZE = I EQ 2)

```

Note that keyword names can be abbreviated to the shortest unambiguous string. The last example sets one of the three keywords according to the value of the variable *I*.

This function example uses the system function `KEYWORD_SET` to determine if a keyword parameter has been passed and if it is nonzero. This is similar to using the condition:

```
IF N_ELEMENTS(P) NE 0 THEN IF P THEN ... ..
```

to test if keywords that have a true/false value are both present and true.

# Keyword Inheritance

Keyword inheritance allows IDL routines to accept keyword parameters not defined in their function or procedure declaration and pass them on to routines they call. This greatly simplifies writing “wrapper” routines, which are variations of a system or user-provided routine. Specifically, keyword inheritance allows your routines to accept keywords accepted by routines that it calls without explicitly handling each keyword individually.

There are two distinct mechanisms to handle keyword inheritance: one to pass keyword parameters by *value*, and another to pass keyword parameters by *reference*.

## **`_EXTRA`: Passing Keyword Parameters by Value**

You can pass keyword parameters to called routines by *value* by adding the formal keyword parameter “`_EXTRA`” (note the underscore character) to the definition of your routine. Passing parameters by value means that you are giving the called routine the *contents* of an existing IDL variable to work with. In turn, this means that keyword parameters passed into a routine by value cannot be returned to the calling routine — there is no variable name into which the value can be placed.

When a routine is defined with the formal keyword parameter `_EXTRA`, pairs of unrecognized keywords and values are placed in an anonymous structure. The name of each unrecognized keyword becomes a tag name, and the keyword value becomes the tag value. Changes to this structure created by using the `_EXTRA` keyword do not affect variables in the calling program.

When the keyword `_EXTRA` appears in a procedure or function call, its argument is either a structure containing additional keyword/value pairs which are inserted into the argument list, or a string array as described in the next section. The value of `_EXTRA` can also be “undefined”, indicating that no additional keyword parameters were passed.

## **`_REF_EXTRA`: Passing Keyword Parameters by Reference**

You can pass keyword parameters to called routines by *reference* by adding the formal keyword parameter “`_REF_EXTRA`” (note the underscore character) to the definition of your routine. Passing parameters by reference means that you are giving the called routine the *name* of an existing IDL variable to work with; IDL takes care of keeping track of the value associated with the name. The *values* of keyword parameters specified via `_REF_EXTRA` are *not* available to the routine that is passing the keywords on.

When a routine is defined with the formal keyword parameter `_REF_EXTRA`, pairs of unrecognized keywords and values are placed in a storage location that is accessible to both calling and called routines, and the keyword names are placed in an IDL string array. The string array can be “deciphered” using the `_EXTRA` keyword, which matches the names in the string with the “live” values in the storage location. This means that if the keywords specify IDL variables, the values of those variables can be altered by any routine that has access to the variable via the keyword inheritance mechanism. In this fashion, the values of keyword parameters can be changed within a routine and passed back to the routine’s caller.

The “pass by reference” keyword inheritance mechanism is especially useful when writing object methods, which may be inherited multiple times and which often wish to change the value of variables available to the calling method. (The values of object properties are one example of data that can profitably be shared by objects at various levels in an object hierarchy.)

## Accepting Extra Keyword Parameters

While you must choose whether a routine will *pass* extra keyword parameters by value or by reference when defining the routine (specifying both `_EXTRA` and `_REF_EXTRA` as formal parameters will cause an error), routines that *accept* extra keyword parameters can use either the `_EXTRA` keyword or the `_REF_EXTRA` keyword. However, it is not possible to both have access to the keyword values and pass them along to called routines by reference within the same routine. This means that any routine that needs access to the passed keyword parameters must use `_EXTRA` in its definition statement, or define the keyword explicitly itself.

### Selective Keyword Redirection

If extra keyword parameters have been passed by reference, you can direct different inherited keywords to different routines by specifying a string or array of strings containing keyword names via the `_EXTRA` keyword. For example, suppose that we write a procedure named `SOMEPROC` that passes extra keywords by reference:

```
PRO SOMEPROC, _REF_EXTRA = ex
ONE, _EXTRA=[ 'MOOSE', 'SQUIRREL' ]
TWO, _EXTRA='SQUIRREL'
END
```

If we call the `SOMEPROC` routine with three keywords:

```
SOMEPROC, MOOSE=moose, SQUIRREL=3, SPY=PTR_NEW(moose)
```

- it will pass the keywords `MOOSE` and `SQUIRREL` and their values (the IDL variable `moose` and the integer 3, respectively) to procedure `ONE`,

- it will pass the keyword SQUIRREL at its value to procedure TWO,
- it will do nothing with the keyword SPY.

## Choosing a Keyword Inheritance Mechanism

The “pass by reference” (`_REF_EXTRA`) keyword inheritance mechanism was introduced in IDL version 5.1, and in many cases is a good choice even if values are not being passed back to the calling routine. Because the `_REF_EXTRA` mechanism does not create an IDL structure to hold the keyword/value pairs, overhead is slightly reduced. Two situations lend themselves to use of the `_REF_EXTRA` mechanism:

1. You need to pass the values of keyword variables back from a called routine to the calling routine.
2. Your routine is an “inner loop” routine that may be called many times. If the routine is called repeatedly, the savings resulting from not creating a new IDL structure with each call may be significant.

It is important to remember that if the routine that is passing the keyword values through also needs access to the values of the keywords for some reason, you must use the “pass by value” (`_EXTRA`) mechanism.

### Note

---

Updating existing routines that use `_EXTRA` to use `_REF_EXTRA` is relatively easy. Since the called routine uses `_EXTRA` to receive the extra keywords in either case, you need only change the `_EXTRA` to `_REF_EXTRA` in the definition of the calling routine.

---

By contrast, the “pass by value” (`_EXTRA`) keyword inheritance mechanism is useful in the following situations:

1. Your routine needs access to the values of the extra keywords for some reason.
2. You want to ensure that variables specified as keyword parameters are not changed by a called routine.

## Example: Keywords Passed by Value

One of the most common uses for the “pass by value” keyword inheritance mechanism is to create “wrapper” routines that extend the functionality of existing routines. In most “wrapper” routines, there is no need to return values to the calling routine — the aim is simply to implement the complete set of keywords available to the existing routine in the wrapper routine.

For example, suppose that procedure TEST is a wrapper to the PLOT command. The text of such a procedure is shown below:

```
PRO TEST, a, b, _EXTRA = e, COLOR = color
    PLOT, a, b, COLOR = color, _EXTRA = e
END
```

The procedure definition:

```
PRO TEST, a, b, _EXTRA = e, COLOR = color
```

places unrecognized keywords (e.g., any keywords other than COLOR) and their values into the variable “e”. If there are no unrecognized keywords, e will be undefined.

When procedure TEST is called with the following command:

```
TEST, x, y, COLOR=3, LINESYLE = 4, THICK=5
```

variable “e”, within TEST, contains an anonymous structure with the value:

```
{ LINESYLE: 4, THICK: 5 }
```

These keyword/value pairs are then be passed from TEST to the PLOT routine using the \_EXTRA keyword:

```
PLOT, a, b, COLOR = color, _EXTRA = e
```

Note that keywords passed into a routine via \_EXTRA override previous settings of that keyword. For example, the call:

```
PLOT, a, b, COLOR = color, _EXTRA = {COLOR: 12}
```

specifies a color index of 12 to PLOT.

## Example: Keywords Passed by Reference

The “pass by reference” keyword inheritance mechanism allows you to change the value of a variable in the calling routine’s context from within the routine. To demonstrate the difference between \_EXTRA and \_REF\_EXTRA, consider the following simple example procedures:

```
PRO TEST1, _EXTRA = ex
HELP, _EXTRA = ex
END

PRO TEST2, _REF_EXTRA = ex
HELP, _EXTRA = ex
END
```

Both TEST1 and TEST2 are simple wrappers to the HELP procedure. Observe the result when we call each routine, specifying OUTPUT as an extra keyword parameter, then use the HELP procedure again to determine the value of the output variable:

```
TEST1, OUTPUT = out & HELP, out
```

IDL prints:

```
% At TEST1                2 /dev/tty
% $MAIN$
EX UNDEFINED = <Undefined>
Compiled Procedures:
    $MAIN$ TEST1
Compiled Functions:
```

Now run TEST2:

```
TEST2, OUTPUT = out & HELP, out
```

IDL prints:

```
OUT                STRING    = Array[8]
```

# Entering Procedure Definitions

Procedures and functions are compiled using the `.RUN` or `.COMPILE` executive commands. The format of these commands is as follows:

```
.RUN File1 [, Filen, ... ]  
.COMPILE File1 [, Filen, ... ]
```

From 1 to 10 files, each containing one or more program units, can be compiled. For more information, see `.RUN` and `.COMPILE` in the *IDL Reference Guide*.

To enter program text directly from the keyboard, simply enter `.RUN` at the `IDL>` prompt. IDL will prompt with the “-” character, indicating that it is compiling a directly entered program. As long as IDL requires more text to complete a program unit, it prompts with the “-” character. Rather than executing statements immediately after they are entered, IDL compiles the program unit as a whole.

Procedure and function definition statements cannot be entered in the single-statement mode, but must be prefaced by either `.RUN` or `.RNEW`.

The first non-empty line the IDL compiler reads determines the type of the program unit: procedure, function, or main program. If the first non-empty line is not a procedure or function definition statement, the program unit is assumed to be a main program. The name of the procedure or function is given by the identifier following the keyword `PRO` or `FUNCTION`. If a program unit with the same name is already compiled, it is replaced by the new program unit.

## Note Regarding Functions

User-defined functions, with the exception of those contained in directories specified by the IDL system variable `!PATH`, must be compiled before the first reference to the function is compiled. This is necessary because the IDL compiler is unable to distinguish between a reference to a variable subscripted with parentheses and a call to a presently undefined user function with the same name. For example, in the statement

```
A = XYZ(5)
```

it is impossible to tell by context alone if `XYZ` is an array or a function.

### Note

In versions of IDL prior to version 5.0, parentheses were used to enclose array subscripts. While using parentheses to enclose array subscripts will continue to work as in previous version of IDL, we strongly suggest that you use brackets in all



new code. See “[Array Subscript Syntax: \[ \] vs. \( \)](#)” on page 157 for additional details.

---

When IDL encounters references that may be either a function call or a subscripted variable, it searches the current directory, then the directories specified by `!PATH`, for files with names that match the unknown function or variable name. If one or more files matching the unknown name exist, IDL compiles them before attempting to evaluate the expression. If no function or variable with the given name exists, IDL displays an error message.

There are several ways to avoid this problem:

- Compile the lowest-level functions (those that call no other functions) first, then higher-level functions, and finally procedures.
- Place the function in a file with the same name as the function, and place that file in one of the directories specified by `!PATH`.
- Use the `FORWARD_FUNCTION` definition statement to inform IDL that a given name refers to a function rather than a variable. See “[Forward Function Definition](#)” on page 217.
- Manually compile all functions before any reference, or use `RESOLVE_ROUTINE` or `RESOLVE_ALL` to compile functions.

# Parameter Passing Mechanism

Parameters are passed to IDL system and user-written procedures and functions by *value* or by *reference*. It is important to recognize the distinction between these two methods.

- Expressions, constants, system variables, and subscripted variable references are passed by value.
- Variables are passed by reference.

Parameters passed by value can only be inputs to program units. Results cannot be passed back to the caller by these parameters. Parameters passed by reference can convey information in either or both directions. For example, consider the following trivial procedure:

```
PRO ADD, A, B
  A = A + B
  RETURN
END
```

This procedure adds its second parameter to the first, returning the result in the first. The call

```
ADD, A, 4
```

adds 4 to A and stores the result in variable A. The first parameter is passed by reference and the second parameter, a constant, is passed by value.

The following call does nothing because a value cannot be stored in the constant 4, which was passed by value.

```
ADD, 4, A
```

No error message is issued. Similarly, if ARR is an array, the call

```
ADD, ARR[5], 4
```

will not achieve the desired effect (adding 4 to element ARR[5]), because subscripted variables are passed by value. The correct, though somewhat awkward, method is as follows:

```
TEMP = ARR[5]
ADD, TEMP, 4
ARR[5] = TEMP
```

**Note**

---

IDL structures behave in two distinct ways. Entire structures are passed by reference, but individual structure fields are passed by value. See [“Parameter Passing with Structures”](#) on page 140 for additional details.

---

# Calling Mechanism

When a user-written procedure or function is called, the following actions occur:

1. All of the actual arguments in the user-procedure call list are evaluated and saved in temporary locations.
2. The actual parameters that were saved are substituted for the formal parameters given in the definition of the called procedure. All other variables local to the called procedure are set to undefined.
3. The function or procedure is executed until a RETURN or RETALL statement is encountered. Procedures also can return on an END statement. The result of a user-written function is passed back to the caller by specifying it as the value of a RETURN statement. RETURN statements in procedures cannot specify a return value.
4. All local variables in the procedure, those variables that are neither parameters nor common variables, are deleted.
5. The new values of the parameters that were passed by reference are copied back into the corresponding variables. Actual parameters that were passed by value are deleted.
6. Control resumes in the calling procedure after the procedure call statement or function reference.

## Recursion

Recursion (i.e., a program calling itself) is supported for both procedures and functions.

## Example

Here is an example of an IDL procedure that reads and plots the next vector from a file. This example illustrates using common variables to store values between calls, as local parameters are destroyed on exit. It assumes that the file containing the data is open on logical unit 1 and that the file contains a number of 512-element, floating-point vectors.

```
;Read and plot the next record from file 1. If RECNO is specified,  
;set the current record to its value and plot it.  
PRO NXT, recno  
  
;Save previous record number.
```

```
COMMON NXT_COM, lastrec

;Set record number if parameter is present.
IF N_PARAMS(0) GE 1 THEN lastrec = recno

;Define LASTREC if this is first call.
IF N_ELEMENTS(lastrec) LE 0 THEN lastrec = 0

;Define file structure.
AA = ASSOC(1, FLTARR(512))

;Read and plot record.
PLOT, AA[lastrec]

;Increment record for next time.
lastrec = lastrec + 1

RETURN A

END
```

Once the user has opened the file, typing `NXT` will read and plot the next record. Typing `NXT, N` will read and plot record number `N`.

## Setting Compilation Options

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

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

where *opt*<sub>*n*</sub> is any of the following:

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

```
COMPILE_OPT DEFINT32, STRICTARR
```

- **DEFINT32** — IDL should assume that lexical integer constants are the 32-bit LONG type rather than the default of 16-bit integers. This takes effect from the point where the `COMPILE_OPT` statement appears in the routine being compiled.
- **HIDDEN** — This routine should not be displayed by `HELP`, unless the `FULL` keyword to `HELP` is used. This directive can be used to hide helper routines that regular IDL users are not interested in seeing.

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

- **OBSOLETE** — If the user has `!WARN.OBS_ROUTINES` set to `True`, attempts to compile a call to this routine will generate warning messages that this routine is obsolete. This directive can be used to warn people that there may be better ways to perform the desired task.
- **STRICTARR** — While compiling this routine, IDL will not allow the use of parenthesis to index arrays, reserving their use only for functions. Square brackets are then the only way to index arrays. Use of this directive will prevent the addition of a new function in future versions of IDL, or new libraries of IDL code from any source, from changing the meaning of your code, and is an especially good idea for library functions.

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

Research Systems recommends the use of

```
COMPILE_OPT IDL2
```

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

```
COMPILE_OPT idl2, HIDDEN
```

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







# Chapter 14: Programming in IDL

The following topics are covered in this chapter:

---

Overview of Programming in IDL . . . . .	306	IDL System Functions and Procedures . . .	319
Informational Routines . . . . .	307	Use Constants of the Correct Type . . . . .	320
Program Control Routines . . . . .	312	Eliminate Invariant Expressions . . . . .	321
Expression Evaluation Order . . . . .	314	Virtual Memory . . . . .	322
Avoid IF Statements . . . . .	315	IDL Implementation . . . . .	328
Use Vector and Array Operations . . . . .	317		

# Overview of Programming in IDL

While IDL is useful as an *ad hoc*, interactive data analysis tool, it is also a powerful programming language. This chapter discusses routines that are useful when writing programs—from simple procedures and functions to large applications—in the IDL language, and presents ideas to consider when trying to create the most efficient programs possible. The routines discussed in this chapter are useful primarily (though not exclusively) in IDL procedures and functions. These routines are rarely used interactively. They provide information about variables and expressions and give the programmer control over program operation.

Routines dealing with error control and handling are discussed in [Chapter 15](#), “Controlling Errors”.

Knowledge of IDL’s implementation and the pitfalls of virtual memory can be used to greatly improve the efficiency of IDL programs. In IDL, complicated computations can be specified at a high level. Therefore, inefficient IDL programs can suffer severe speed penalties — perhaps much more so than with most other languages.

Techniques for writing efficient programs in IDL are identical to those in other computer languages with the addition of the following simple guidelines:

- Use array operations rather than loops wherever possible. Try to avoid loops with high repetition counts.
- Use IDL system functions and procedures wherever possible.
- Access array data in machine address order.

Attention also must be given to algorithm complexity and efficiency, as this is usually the greatest determinant of resources used.

# Informational Routines

Informational routines return information about variables, expressions, parameters, etc. The routines `KEYWORD_SET`, `N_ELEMENTS`, `N_PARAMS`, and `SIZE` are useful in procedures and functions to check if arguments are supplied. Procedures should be written to check that all required arguments are supplied and to supply reasonable default values for missing optional parameters. In addition, `TAG_NAMES` and `N_TAGS`, which are discussed in “Advanced Structure Usage” on page 147, supply information about structure variables.

## ARG\_PRESENT Function

The `ARG_PRESENT` function returns `TRUE` if its parameter will be passed back to the caller. This function is useful in user-written procedures to determine if a created value remains within the scope of the calling routine. `ARG_PRESENT` helps the caller avoid expensive computations and prevents heap leaks. For example, assume that a procedure exists which depends upon an argument passed by the caller:

```
PRO pass_it, i
```

If the caller does not specify *i*, the program may not function properly. You can check to make sure that an argument was specified by using the following statement:

```
IF ARG_PRESENT(i) THEN BEGIN
```

## KEYWORD\_SET Function

The `KEYWORD_SET` function returns a 1 (true), if its parameter is defined and nonzero; otherwise, it returns zero (false). For example, assume that a procedure is written which performs and returns the result of a computation. If the keyword `PLOT` is present and nonzero, the procedure also plots its result as follows:

```
;Procedure definition.
PRO XYZ, result, PLOT = plot

;Compute result.
...

;Plot result if keyword parameter is set.
IF KEYWORD_SET(PLOT) THEN PLOT, result

END
```

A call to this procedure that produces a plot is shown in the following statement.

```
XYZ, R, /PLOT
```

## N\_ELEMENTS Function

The `N_ELEMENTS` function returns the number of elements contained in any expression or variable. Scalars always have one element. The number of elements in arrays or vectors is equal to the product of the dimensions. The `N_ELEMENTS` function returns zero if its parameter is an undefined variable. The result is always a longword scalar.

For example, the following expression is equal to the mean of a numeric vector or array.

```
TOTAL(arr) / N_ELEMENTS(arr)
```

The `N_ELEMENTS` function provides a convenient method of determining if a variable is defined. The following statement sets the variable `abc` to zero if it is undefined; otherwise, the variable is not changed.

```
IF N_ELEMENTS(abc) EQ 0 THEN abc = 0
```

`N_ELEMENTS` is frequently used to check for omitted plain and keyword arguments. `N_PARAMS` cannot be used to check for the number of keyword arguments because it returns only the number of plain arguments. An example of using `N_ELEMENTS` to check for a keyword parameter is as follows:

```
;Display an image with a given zoom factor. If factor is omitted,
;use 4.
PRO ZOOM, image, FACTOR = factor

;Supply default for missing keyword parameter.
IF N_ELEMENTS(factor) EQ 0 THEN factor = 4
```

### Note

If you use this method, the variable `factor` is defined having the value 4, even though no value was supplied by the user. If the `ZOOM` procedure were called within another routine, the variable `factor` would be defined for that routine and for any other routines also called by the routine that called `ZOOM`. This can lead to unexpected behavior if you pass arguments from one routine to another.

You can avoid this problem by using different variable names inside the routine than are used in calling the routine. For example, if you wanted to supply a default zoom factor in the example above, but did not want to change the value of `factor`, you could use an approach similar to the following:

```
IF N_ELEMENTS(factor) EQ 0 THEN zoomfactor = 4 $
ELSE zoomfactor = factor
```

You would then set the zoom factor internally using the `zoomfactor` variable, leaving `factor` itself unchanged.

---

## N\_PARAMS Function

The `N_PARAMS` function returns the number of positional arguments (not keyword arguments) present in a procedure or function call. A frequent use is to call `N_PARAMS` to determine if all arguments are present and if not, to supply default values for missing parameters. For example:

```

;Print values of XX and YY. If XX is omitted, print values of YY
;versus element number.
PRO XPRINT, XX, YY

    ;Check number of arguments.
    CASE N_PARAMS() OF

        ;Single-argument case.
        1: BEGIN

            ;First argument is y values.
            Y = XX

            ;Create vector of subscript indices.
            X = INDGEN(N_ELEMENTS(Y))

            END

        ;Two-argument case.
        2: BEGIN

            ;Copy parameters to local arguments.
            Y = YY & X = XX

            END

        ;Print error message.
        ELSE: MESSAGE, 'Wrong number of arguments'

    ENDCASE

    ;Remainder of procedure.
    ...

END

```

## SIZE Function

The `SIZE` function returns a vector that contains information indicating the size and type of the parameter. The returned vector is always of longword type. The first element is equal to the number of dimensions of the parameter and is zero if the parameter is a scalar. The following elements contain the size of each dimension. After the dimension sizes, the last two elements indicate the type of the parameter and the total number of elements, respectively. The type is encoded as follows:

Type Code	Data Type
0	Undefined
1	Byte
2	Integer (16-bit)
3	Longword integer (32-bit)
4	Floating point
5	Double-precision floating
6	Complex floating
7	String
8	Structure
9	Double-precision complex floating
10	Pointer
11	Object reference
12	Unsigned integer (16-bit)
13	Unsigned longword integer (32-bit)
14	64-bit integer
15	Unsigned 64-bit integer

*Table 14-1: Type Codes Returned by the SIZE Function*

## Examples

Assume `A` is an integer array with dimensions of (3,4,5). The statements:

```
arr = INDGEN(3,4,5)
```

```
S = SIZE(arr)
```

assigns to the variable, *S*, a six-element vector containing:

Element	Value	Description
$S_0$	3	Three dimensions
$S_1$	3	First dimension
$S_2$	4	Second dimension
$S_3$	5	Third dimension
$S_4$	2	Integer type
$S_5$	60	Number of elements = 3*4*5

*Table 14-2: SIZE Values*

A code segment that checks to see if a variable (*arr*) is two-dimensional and extracts the dimensions is shown below.

```
;Create a variable.
arr = [[1,2,3],[4,5,6]]

;Get size vector.
S = SIZE(arr)

;Check if two dimensional.
IF S[0] NE 2 THEN $
    ;Print error message.
    MESSAGE, 'Variable a is not two dimensional.'

;Get number of columns and rows.
NX = S[1] & NY = S[2]

PRINT, 'Array is ', NX, ' columns by ', NY, ' rows.'
```

# Program Control Routines

IDL provides the following routines to control the flow of an IDL program: `CALL_FUNCTION`, `CALL_PROCEDURE`, `EXECUTE`, `EXIT`, `STOP`, and `WAIT`.

The program control procedures are largely self-explanatory with the exception of the `EXECUTE` function. `CALL_FUNCTION` and `CALL_PROCEDURE` are used to indirectly call functions and procedures whose names are contained in strings. The `EXIT` procedure exits the IDL session. `STOP` halts execution of a program or batch file and prints the values of its optional parameters. `WAIT`, as its name implies, pauses execution for a given amount of time specified in seconds.

## CALL\_FUNCTION and CALL\_PROCEDURE

`CALL_FUNCTION` and `CALL_PROCEDURE` are used to indirectly call functions and procedures whose names are contained in strings.

`CALL_FUNCTION` calls the IDL function specified, passing any additional parameters as its arguments. The result of the called function is passed back as the result of the routine. `CALL_PROCEDURE` calls the procedure, passing any additional parameters as its arguments.

Although not as flexible as the `EXECUTE` function, `CALL_FUNCTION` and `CALL_PROCEDURE` are much faster, and should be used in preference to `EXECUTE` whenever possible.

## EXECUTE

The `EXECUTE` function compiles and executes one or more IDL statements contained in its string parameter during runtime.

`EXECUTE` is limited by two factors:

- Calls to `EXECUTE` cannot be nested, so a routine called by `EXECUTE` cannot use `EXECUTE` itself.
- The need to compile the string at runtime makes `EXECUTE` inefficient in terms of speed.

The `CALL_FUNCTION` and `CALL_PROCEDURE` routines provide much of the functionality of `EXECUTE` without imposing these limitations and should be used in preference to `EXECUTE` when possible.



The result of the EXECUTE function is true (1) if the string was successfully compiled and executed. If an error occurred during either phase, the result is false (0). If an error occurs, an error message is printed.

Multiple statements in the string should be separated with the “&” character. GOTO statements and labels are not allowed.

### Example

This example code fragment, taken from the routine SVDFIT, calls a function whose name is passed to SVDFIT via a keyword parameter as a string. If the keyword parameter is omitted, the function POLY is called.

```

;Function declaration.
FUNCTION SVDFIT,..., FUNCT = funct

...

;Use default name, POLY, for function if not specified.
IF N_ELEMENTS(FUNCT) EQ 0 THEN FUNCT = 'POLY'

;Make a string of the form "a = funct(x,m)", and execute it.
Z = EXECUTE('A = '+FUNCT+'(X,M)')

...

```

The above example is easily made more efficient by replacing the call to EXECUTE with the following line:

```
A = CALL_FUNCTION(FUNCT, X, M)
```

## EXIT

The **EXIT** procedure quits IDL and exits back to the operating system. All buffers are flushed and open files are closed. The values of all variables that were not saved are lost.

## STOP

The **STOP** procedure stops the execution of a running program or batch file. Control reverts to the interactive mode.

## WAIT

The **WAIT** procedure suspends execution of an IDL program for a specified period. Note that because of other activity on the system, the duration of program suspension may be longer than requested.

## Expression Evaluation Order

The order in which an expression is evaluated can have a significant effect on program speed. Consider the following statement, where A is an array:

```
;Scale A from 0 to 16.  
B = A * 16. / MAX(A)
```

This statement first multiplies every element in A by 16 and then divides each element by the value of the maximum element. The number of operations required is twice the number of elements in A. A much faster way of computing the same result is used in the following statement:

```
;Scale A from 0 to 16 using only one array operation.  
B = A * (16./MAX(A))
```

or

```
;Operators of equal priority are evaluated from left to right. Only  
;one array operation is required.  
B = 16./MAX(A) * A
```

The faster method only performs one operation for each element in A, plus one scalar division. To see the speed difference on your own machine, execute the following statements:

```
A = RANDOMU(seed, 512, 512)  
t1 = SYSTIME(1) & B = A*16./MAX(A) & t2 = SYSTIME(1)  
PRINT, 'Time for inefficient calculation: ', t2-t1  
t3 = SYSTIME(1) & B = 16./MAX(A)*A & t4 = SYSTIME(1)  
PRINT, 'Time for efficient calculation: ', t4-t3
```

# Avoid IF Statements

Programs with array expressions run faster than programs with scalars, loops, and IF statements. Some examples of slow and fast ways to achieve the same results follow.

## Example—Summing Elements

The first example adds all positive elements of array B to array A.

```
;Using a loop will be slow.
FOR I = 0, (N-1) DO IF B[I] GT 0 THEN A[I] = A[I] + B[I]

;Fast way: Mask out negative elements using array operations.
A = A + (B GT 0) * B

;Faster way: Add B > 0.
A = A + (B > 0)
```

When an IF statement appears in the middle of a loop with each element of an array in the conditional, the loop can often be eliminated by using logical array expressions.

## Example—Using Array Operators and the WHERE Function

In the example below, each element of C is set to the square-root of A if A[I] is positive; otherwise, C[I] is set to minus the square-root of the absolute value of A[I].

```
;Using an IF statement is slow.
FOR I=0,(N-1) DO IF A[I] LE 0 THEN $
    C[I]=SQRT(-A[I]) ELSE C[I]=SQRT(A[I])

;Using an array expression is much faster.
C = ((A GT 0) * 2 - 1) * SQRT(ABS(A))
```

The expression (A GT 0) has the value 1 if A[I] is positive and has the value 0 if A[I] is not. (A GT 0)\* 2 - 1 is equal to +1 if A[I] is positive or -1 if A[I] is negative, accomplishing the desired result without resorting to loops or IF statements.

Another method is to use the WHERE function to determine the subscripts of the negative elements of A and negate the corresponding elements of the result.

```
;Get subscripts of negative elements.
negs = WHERE(A LT 0)

;Take root of absolute value.
C = SQRT(ABS(A))
```

```
;Negate elements in C corresponding to negative elements in A.  
C[negs] = -C[negs]
```

## Use Vector and Array Operations

Whenever possible, vector and array data should always be processed with IDL array operations instead of scalar operations in a loop. For example, consider the problem of inverting a  $512 \times 512$  image. This problem arises because approximately half the available image display devices consider the origin to be the lower-left corner of the screen, while the other half recognize it as the upper-left corner.

The following example is for demonstration only. The IDL system variable `!ORDER` should be used to control the origin of image devices. The `ORDER` keyword to the `TV` procedure serves the same purpose.

A programmer without experience in using IDL might be tempted to write the following nested loop structure to solve this problem:

```
FOR I = 0, 511 DO FOR J = 0, 255 DO BEGIN

    ;Temporarily save pixel image.
    temp = image[I, J]

    ;Exchange pixel in same column from corresponding row at bottom
    image[I, J] = image[I, 511 - J]

    image[I, 511-J] = temp

ENDFOR
```

A more efficient approach to this problem capitalizes on IDL's ability to process arrays as a single entity:

```
FOR J = 0, 255 DO BEGIN

    ;Temporarily save current row.
    temp = image[*, J]

    ;Exchange row with corresponding row at bottom.
    image[*, J] = image[*, 511-J]

    image[*, 511-J] = temp

ENDFOR
```

At the cost of using twice as much memory, processing can be simplified even further by using the following statements:

```
;Get a second array to hold inverted copy.
image2 = BYTARR(512, 512)
```

```
;Copy the rows from the bottom up.  
FOR J = 0, 511 DO image2[* , J] = image[* , 511-J]
```

Even more efficient is the single line:

```
image2 = image[* , 511 - INDGEN(512)]
```

that reverses the array using subscript ranges and array-valued subscripts.

Finally, using the built-in ROTATE function is quickest of all:

```
image = ROTATE(image, 7)
```

Inverting the image is equivalent to transposing it and rotating it 270 degrees clockwise.

# IDL System Functions and Procedures

IDL supplies a number of built-in functions and procedures to perform common operations. These system-supplied functions have been carefully optimized and are almost always much faster than writing the equivalent operation in IDL with loops and subscripting.

## Example

A common operation is to find the sum of the elements in an array or subarray. The **TOTAL** function directly and efficiently evaluates this sum at least 10 times faster than directly coding the sum.

```
;Slow way: Initialize SUM and sum each element.  
sum = 0. & FOR I = J, K DO sum = sum + array[I]  
  
;Efficient, simple way.  
sum = TOTAL(array[J:K])
```

Similar savings result when finding the minimum and maximum elements in an array (**MIN** and **MAX** functions), sorting (**SORT** function), finding zero or nonzero elements (**WHERE** function), etc.

## Use Constants of the Correct Type

As explained in [Chapter 5, “Constants”](#), the syntax of a constant determines its type. Efficiency is adversely affected when the type of a constant must be converted during expression evaluation. Consider the following expression:

$$A + 5$$

If the variable *A* is of floating-point type, the constant 5 must be converted from short integer type to floating point each time the expression is evaluated.

The type of a constant also has an important effect in array expressions. Care must be taken to write constants of the correct type. In particular, when performing arithmetic on byte arrays with the intent of obtaining byte results, be sure to use byte constants; e.g., *nB*. For example, if *A* is a byte array, the result of the expression *A + 5B* is a byte array, while *A + 5* yields a 16-bit integer array.



## Eliminate Invariant Expressions

Expressions whose values do not change inside a loop should be moved outside the loop. For example, in the loop:

```
FOR I = 0, N - 1 DO arr[I, 2*J-1] = ...,
```

the expression  $(2*J-1)$  is invariant and should be evaluated only once before the loop is entered:

```
temp = 2*J-1  
FOR I = 0, N-1 DO arr[I, temp] = ....
```

# Virtual Memory

The IDL programmer and user must be cognizant of the characteristics of virtual memory computer systems to avoid penalty. Virtual memory allows the computer to execute programs that require more memory than is actually present in the machine by keeping those portions of programs and data that are not being used on the disk. Although this process is transparent to the user, it greatly affects the efficiency of the program.

IDL arrays are stored in dynamically allocated memory. Although the program can address large amounts of data, only a small portion of that data actually resides in physical memory at any given moment; the remainder is stored on disk. The portion of data and program code in real physical memory is commonly called the working set.

When an attempt is made to access a datum in virtual memory not currently residing in physical memory, the operating system suspends IDL, arranges for the page of memory containing the datum to be moved into physical memory and then allows IDL to continue. This process involves deciding where the datum should go in memory, writing the current contents of the selected memory page out to the disk, and reading the page with the datum into the selected memory page. A *page fault* is said to occur each time this process takes place. Because the time required to read from or write to the disk is very large in relation to the physical memory access time, page faults become an important consideration.

When using IDL with large arrays, it is important to have access to sufficient physical and virtual memory. Given a suitable amount of physical memory, the parameters that regulate virtual memory require adjustment to assure best performance. These parameters are discussed below. See “[Virtual Memory System Parameters](#)” on page 325. If you suspect that lack of physical or virtual memory is causing problems, consult your system manager.

## Access Large Arrays by Memory Order

When an array is larger than or close to the working set size (i.e., the amount of physical memory available for the process), it is preferable to access it in memory address order.

Consider the process of transposing a large array. Assume the array is a  $512 \times 512$  byte image with a 100 kilobyte working set. The array requires  $512 \times 512$ , or approximately 250 kilobytes. Less than half of the image can be in memory at any one instant.

In the transpose operation, each row must be interchanged with the corresponding column. The first row, containing the first 512 bytes of the image, will be read into memory, if necessary, and written to the first column. Because arrays are stored in row order (the first subscript varies the fastest), one column of the image spans a range of addresses almost equal to the size of the entire image. To write the first column, 250,000 bytes of data must be read into physical memory, updated, and written back to the disk. This process must be repeated for each column, requiring the entire array be read and written almost 512 times. The amount of time required to transpose the array using the method described above is relatively large.

In contrast, the IDL `TRANPOSE` function transposes large arrays by dividing them into subarrays smaller than the working set size enabling it to transpose a  $512 \times 512$  image in a much smaller amount of time.

### Example

Consider the operation of the following IDL statement:

```
FOR X = 0, 511 DO FOR Y = 0, 511 DO ARR[X, Y] = ...
```

This statement requires an extremely large execution time because the entire array must be transferred between memory and the disk 512 times. The proper form of the statement is to process the points in address order by using the following statement:

```
FOR Y = 0, 511 DO FOR X = 0, 511 DO ARR[X, Y] = ...
```

This approach cuts computing time by a factor of at least 50.

## Running Out of Virtual Memory

If you process large images with IDL and use the vendor-supplied default system parameters (especially if you have a small system), you may encounter the error message

```
% Unable to allocate memory.
```

This error message means that IDL was unable to obtain enough virtual memory to hold all your data. Whenever you define an array, image, or vector, IDL asks the operating system for some virtual memory in which to store the data. When you reassign the variable, IDL frees the memory for re-use.

The first time you get this error, you will either have to stop what you are doing and exit IDL or delete unused variables containing images or arrays, thereby releasing enough virtual memory to continue. You can delete the memory allocation of array variables by setting the variable equal to a scalar value.

If you need to exit IDL, you first should use the SAVE procedure to save your variables in an IDL save file. Later, you will be able to recover those variables from the save file using the RESTORE procedure.

The HELP/MEMORY command tells you how much virtual memory you have allocated. For example, a  $512 \times 512$  complex floating array requires  $8 \times 512^2$  bytes or about 2 megabytes of virtual memory because each complex element requires 8 bytes. Deleting a variable containing a  $512 \times 512$  complex array will increase the amount of virtual memory available by this amount.

## Minimizing Virtual Memory

If virtual memory is a problem, try to tailor your programming to minimize the number of images held in IDL variables. Keep in mind that IDL creates temporary arrays to evaluate expressions involving arrays. For example, when evaluating the statement

$$A = (B + C) * (E + F)$$

IDL first evaluates the expression  $B + C$  and creates a temporary array if either B or C are arrays. In the same manner, another temporary array is created if either E or F are arrays. Finally, the result is computed, the previous contents of A are deleted, and the temporary area holding the result is saved as variable A. Therefore, during the evaluation of this statement, enough virtual memory to hold two arrays' worth of data is required in addition to normal variable storage.

It is a good idea to delete the allocation of a variable that contains an image and that appears on the left side of an assignment statement, as shown in the following program.

```

;Loop to process an image.
FOR I = ... DO BEGIN

;Processing steps.
...

;Delete old allocation for A.
A = 0

;Compute image expression and store.
A = Image_Expression

...

;End of loop.
ENDFOR

```

The purpose of the statement `A=0` is to free the old memory allocation for the variable `A` before computing the image expression in the next statement. Because the old value of `A` is going to be replaced in the next statement, it makes sense to free `A`'s allocation first.

## The TEMPORARY Function

Another way to minimize memory use when performing operations on large arrays is to use the `TEMPORARY` function. `TEMPORARY` returns the value of its argument as a temporary variable and makes the argument undefined. In this way, you avoid making a new copy of temporary results. For example, assume that `A` is a large array. To add 1 to each element in `A`, you could enter:

```
A = A+1
```

However, this statement creates a new array for the result of the addition and assigns the result to `A` *before* freeing the old allocation of `A`. Hence, the total storage required for the operation is twice the size of `A`. The statement:

```
A = TEMPORARY(A) + 1
```

requires no additional space.

## Virtual Memory System Parameters

The first step is to determine how much virtual memory you require. For example, if you compute complex Fast Fourier Transforms (FFT) on  $512 \times 512$  images, each complex image requires 2 megabytes. Suppose that during a typical session you need to have four images stored in variables and require enough memory for two images to hold temporary results, resulting in a total of six images or 12 megabytes. Rounding up to 16 megabytes gives a reasonable goal. The following `SYSGEN` parameters and quotas should be changed to increase the amount of virtual memory available.

### Note

---

For UNIX, The size of the swapping area(s) determines how much virtual memory your process is allowed. To increase the amount of available virtual memory, you must increase the size of the swap device (sometimes called the swap partition). Increasing the size of a swap partition is a time-consuming task that should be planned carefully. It usually requires saving the contents of the disk, reformatting the disk with the new file partition sizes, and restoring the original contents. Some systems offer the alternative of swapping to a regular file. This is a considerably easier solution, although it may not be as efficient. Consult your system documentation for details and instructions on how to perform these operations.

---

**Note**

For OpenVMS, as it comes from DEC, is not tuned for image processing. To get the best performance from IDL, you should increase the VMS `SYSGEN` parameters, file sizes, and `AUTHORIZE` quotas that restrict the virtual memory system. This discussion is on the most elementary level, and the appropriate VMS manuals should be consulted for more detail.

**SYSGEN Parameters**

**WSMAX:** This parameter sets the maximum number of pages of any working set on a system-wide basis. The working set is that portion of virtual memory used by a process that is actually in physical memory. Although this is an over simplification, small working set sizes cause page faulting. Page faults waste time and potentially require disk accesses. Increasing the working set to a size of three times the size of the largest array to be processed, or at least 2,000 blocks, can cause dramatic speed improvements.

**VIRTUALPAGECNT:** This parameter sets the maximum number of virtual pages (512 bytes/page) that can be used by any one process.

To change the values of `SYSGEN` parameters, DEC recommends that you run the `AUTOGEN` command procedure after adding lines to set the new values of changed parameters to the end of the file `SYS$SYSTEM: MODPARAMS.DAT`.

**System Files**

The sizes of the system page and swap files (`SYS$SYSTEM: PAGEFILE.SYS` and `SWAPFILE.SYS`) must be large enough to contain the virtual memory used by all active processes. In any event, you cannot have more virtual memory than will fit in the page file. You can increase the size of these files or create secondary system files on a disk other than the system disk. If you get the error message

```
Page file fragmented - continuing
```

on the system console, your page file is too small. To increase the size of these files, use the command procedure `SYS$UPDATE: SWAPFILES`. Use the `SYSGEN INSTALL` command to activate system files created on disks other than the system disk. `AUTOGEN` can also be used to change the sizes of these files.

**Quotas**

The following quotas, all of which can be changed on a per user or system basis using the `AUTHORIZE` utility, affect virtual page limits and working set sizes.

**Pgflquo:** The page file quota for each user expressed in blocks. If you increase the size of the page file, be sure to increase the page file quotas for the users requiring more virtual memory. Be sure that the page file size is at least as large as the sum of the quotas of each active user.

**WSquo:** The working set quota for each user. This quota can be used to allow some users a larger working set than others. *WSquo* must not be larger than *WSMAX*.

**Note** \_\_\_\_\_

For Windows and Macintosh, consult your system documentation for details on how to configure your system to use virtual memory.

---

# IDL Implementation

IDL programs are compiled into a low-level abstract machine code which is interpretively executed. The dynamic nature of variables in IDL and the relative complexity of the operators precludes the use of directly executable code. Statements are only compiled once, regardless of the frequency of their execution.

The IDL interpreter emulates a simple stack machine with approximately 50 operation codes. When performing an operation, the interpreter must determine the type and structure of each operand and branch to the appropriate routine. The time required to properly dispatch each operation may be longer than the time required for the operation itself.

The characteristics of the time required for array operations is similar to that of vector computers and array processors. There is an initial set-up time, followed by rapid evaluation of the operation for each element. The time required per element is shorter in longer arrays because the cost of this initial set-up period is spread over more elements. The speed of IDL is comparable to that of optimized FORTRAN for array operations. When data are treated as scalars, IDL efficiency degrades by a factor of 30 or more.





# Chapter 15: Controlling Errors

The following topics are covered in this chapter:

---

Overview .....	330	Controlling Input/Output Errors .....	338
Default Error-Handling Mechanism .....	331	Error Signaling .....	340
Disappearing Variables .....	332	Obtaining Traceback Information .....	342
Controlling Errors Using CATCH .....	333	Error Handling .....	343
Controlling Errors Using ON_ERROR ...	337	Math Errors .....	345

## Overview

This chapter discusses routines and methods used to check and handle errors that occur in IDL programs. The routines covered here are rarely used interactively.

IDL divides possible execution errors into three categories: input/output, math, and all others. There are three main error-handling routines: `CATCH`, `ON_ERROR`, and `ON_IOERROR`. `CATCH` is a generalized mechanism for handling exceptions and errors. The `ON_ERROR` routine handles regular errors when an error handler established by the `CATCH` procedure is not present. The `ON_IOERROR` routine allows you to change the default way in which input/output errors are handled. The `FINITE` and `CHECK_MATH` routines provide control over math errors.

## Default Error-Handling Mechanism

In the default case, whenever an error is detected by IDL during the execution of a program, program execution stops and an error message is printed. The execution context is that of the program unit (procedure, function, or main program) in which the error occurred.

Sometimes it is possible to recover from an error by manually entering statements to correct the problem. Possibilities include setting the values of variables, closing files, etc., and then entering the command `.CONTINUE`, which resumes execution of the program unit at the beginning of the statement that caused the error.

As an example, if an error occurs because an undefined variable is referenced, you can simply define the variable from the keyboard, then continue execution with `.CON`. Of course, this is a temporary solution. You should still edit the program file to fix the problem permanently.

## Disappearing Variables

IDL users may find that all their variables have seemingly disappeared after an error occurs inside a procedure or function. The misunderstood subtlety is that after the error occurs, IDL's context is *inside the called procedure*, not in the main level. All variables in procedures and functions, with the exception of parameters and common variables, are local in scope. Typing [RETURN](#) or [RETALL](#) will make the lost variables reappear.

[RETALL](#) is best suited for use when an error is detected in a procedure and it is desired to return immediately to the main program level despite nested procedure calls. [RETALL](#) issues [RETURN](#) commands until the main program level is reached.

The [HELP](#) command can be used to see the current call stack (i.e., which program unit IDL is in and which program unit called it). For more information, see [HELP](#) in the *IDL Reference Guide*.

# Controlling Errors Using CATCH

The [CATCH](#) procedure provides a generalized mechanism for handling any type of errors and exceptions within IDL. Calling [CATCH](#) establishes an error handler for the current procedure that intercepts all errors that can be handled by IDL, with the exception of non-fatal warnings such as math errors (e.g., floating-point underflow). The [CATCH](#) mechanism is similar to C's `set jmp/longjmp` facilities or C++'s `catch/throw` facilities.

When an error occurs, each active procedure, beginning with the offending procedure and proceeding up the call stack to the main program level, is examined for an error handler (established by a call to [CATCH](#)). If an error handler is found, control resumes at the statement after the call to [CATCH](#). The index of the error is returned in the argument to [CATCH](#) and is also stored in `!ERROR_STATE.CODE`. The associated error message is stored in `!ERROR_STATE.MSG`. If no error handlers are found, program execution stops, an error message is issued, and control reverts to the interactive mode.

For more information, see [CATCH](#) and [!ERROR\\_STATE](#) in the *IDL Reference Guide*.

## Interaction of CATCH, ON\_ERROR, and ON\_IOERROR

Error handlers established by calls to [CATCH](#) supersede calls to [ON\\_ERROR](#). However, calls to [ON\\_IOERROR](#) made in the procedure that causes an I/O error supersede any error handling mechanisms created with [CATCH](#) and the program

branches to the label specified by ON\_IOERROR. The following figure is a flow chart of how errors are handled in IDL.

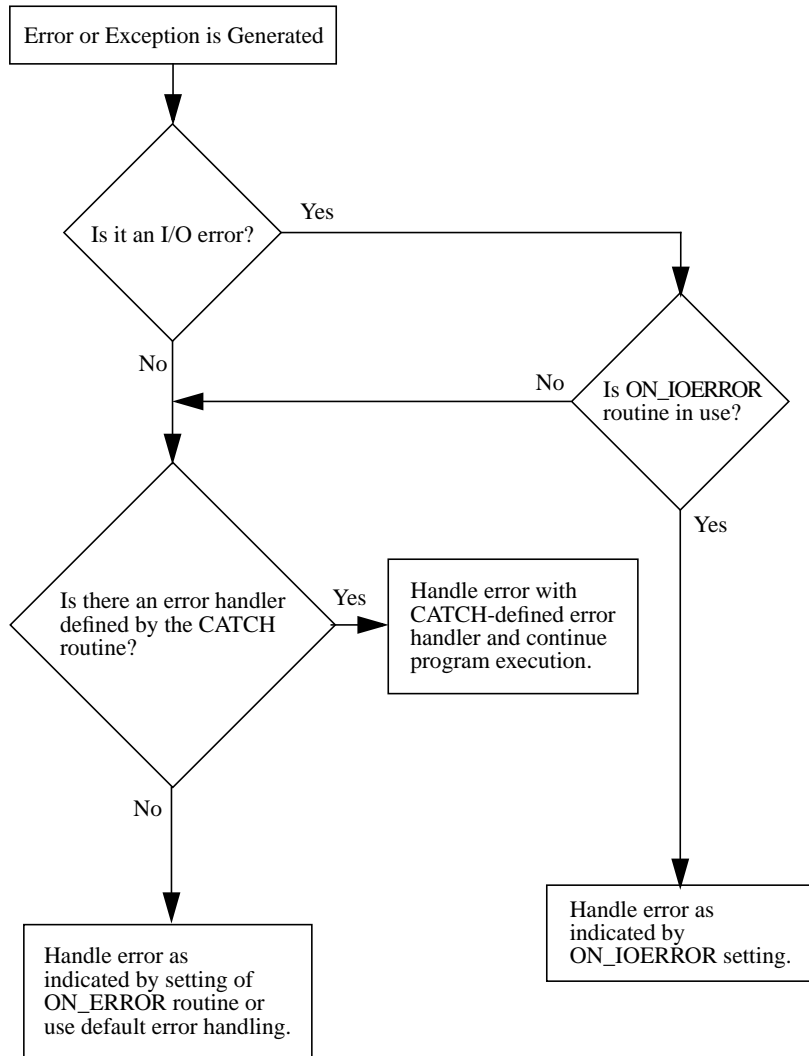


Figure 15-1: Error Handling in IDL.

## Canceling an Error Handler

Call `CATCH` with the `CANCEL` keyword set to cancel a procedure's error handler. This cancellation does not effect other error handlers that may be established in other active procedures.

## Generating an Exception

To generate an exception and cause control to return to the error handler, use the `MESSAGE` procedure. Calling `MESSAGE` generates an exception that sets the `!ERROR` system variable to -5 and the `!ERR_STRING` system variable to the string used as an argument to `MESSAGE`. See [“Error Signaling”](#) on page 340.

## Example Using `CATCH`

The following procedure illustrates the use of `CATCH`:

```

PRO ABC

;Define variable A.
A = FLTARR(10)

;Establish error handler. When errors occur, the index of the error
;is returned in the variable Error_status. Initially, this
;argument is set to zero.
CATCH, Error_status

;This statement begins the error handler.
IF Error_status NE 0 THEN BEGIN

    PRINT, 'Error index: ', Error_status
    PRINT, 'Error message:', !ERR_STRING

    ;Handle the error by extending A.
    A=FLTARR(12)

ENDIF

;Cause an error.
A[11]=12

;Even though an error occurs in the line above, program execution
;continues to this point because the event handler extended the
;definition of A so that the statement can be re-executed.
HELP, A

END

```

Running the ABC procedure causes IDL to produce the following output and control returns to the interactive prompt:

```
Error index:          -101
Error message:
Attempt to subscript A with <INT (      11)> is out of range.
A                FLOAT      = Array(12)
```



# Controlling Errors Using ON\_ERROR

The `ON_ERROR` procedure determines the action taken when an error is detected inside a user procedure or function and no error handlers established with the `CATCH` procedure are found. The possible options for error recovery are shown in the following table:

Value	Action
0	Stop immediately in the context of the procedure or function that caused the error. This is the default action.
1	Return to the main program level and stop.
2	Return to the caller of the program unit that called <code>ON_ERROR</code> and stop.
3	Return to the program unit that called <code>ON_ERROR</code> and stop.

*Table 15-1: Error Recovery Options*

One useful option is to use `ON_ERROR` to cause control to be returned to the caller of a procedure in the event of an error. The statement:

```
ON_ERROR, 2
```

placed at the beginning of a procedure will have this effect. Include this statement in library procedures and other routines that will be used by others once the routines have been debugged. This form of error recovery makes debugging a routine difficult because the routine is exited as soon as an error occurs; therefore, it should be added once the code is completely tested.

Note that error handlers established by `CATCH` supersede calls to `ON_ERROR` made in the same procedure.

# Controlling Input/Output Errors

The default action for handling input/output errors is to treat them exactly like regular errors and follow the error handling strategy set by `ON_ERROR`. You can alter this default by using the `ON_IOERROR` procedure to specify the label of a statement to which execution should jump if an input/output error occurs. When IDL detects an input/output error and an error-handling statement has been established, control passes directly to the given statement without stopping program execution. In this case, no error messages are printed.

Note that calls to `ON_IOERROR` made in the procedure that causes an I/O error supersede any error handling mechanisms created with `CATCH` and the program branches to the label specified by `ON_IOERROR`.

When writing procedures and functions that are to be used by others, it is good practice to anticipate and handle errors caused by the user. For example, the following procedure segment, which opens a file specified by the user, handles the case of a nonexistent file or read error.

```

;Define a function to read, and return a 100-element, floating-
;point array.
FUNCTION READ_DATA, FILE_NAME

;Declare error label.
ON_IOERROR, BAD

;Use the GET_LUN keyword to allocate a logical file unit.
OPENR, UNIT, FILE NAME, /GET_LUN

;Define data array.
A = FLTARR(100)

;Read it.
READU, UNIT, A

;Clean up and return.
GOTO, DONE

;Exception label. Print the error message.
BAD: PRINT, !ERR_STRING

;Close and free the input/output unit.
DONE: FREE_LUN, UNIT

;Return the result. This will be undefined if an error occurred.
RETURN, A

```

END

The important things to note in this example are that the `FREE_LUN` procedure is always called, even in the event of an error, and that this procedure always returns to its caller. It returns an undefined value if an error occurs, causing its caller to encounter the error.

# Error Signaling

The `MESSAGE` procedure is used by user procedures and functions to issue errors. It has the form:

```
MESSAGE, Text
```

where *Text* is a scalar string that contains the text of the error message.

The `MESSAGE` procedure issues error and informational messages using the same mechanism employed by built-in IDL routines. By default, the message is issued as an error, the message is output, and IDL takes the action specified by the `ON_ERROR` procedure.

As a side effect of issuing the error, appropriate fields of the system variable `!ERROR_STATE` are set; the text of the error message is placed in `!ERROR_STATE.MSG`, or in `!ERROR_STATE.SYS_MSG` for the operating system's component of the error message. See “[Error Handling](#)” on page 343 or `!ERROR_STATE` in the *IDL Reference Guide* for more information.

As an example, assume the statement:

```
MESSAGE, 'Unexpected value encountered.'
```

is executed in a procedure named `CALC`. IDL would print:

```
% CALC: Unexpected value encountered.
```

and execution would halt.

The `MESSAGE` procedure accepts several keywords that modify its behavior. See `MESSAGE` in the *IDL Reference Guide* for additional details.

Another use of `MESSAGE` involves re-signaling trapped errors. For example, the following code uses `ON_IOERROR` to read from a file until an error (presumably end-of-file) occurs. It then closes the file and reissues the error.

```
;Open the data file.
OPENR, UNIT, 'DATA.DAT', /GET_LUN

;Arrange for jump to label EOD when an input/output error occurs.
ON_IOERROR, EOD

;Read every line of the file.
WHILE 1 DO READF, UNIT, LINE

;An error has occurred. Cancel the input/output error trap.
EOD: ON_IOERROR, NULL
```

```
;Close the file.  
FREE_LUN, UNIT  
  
;Reissue the error. !ERR_STRING contains the appropriate text. The  
;IOERROR keyword causes it to be issued as an input/output error.  
;Use of NONAME prevents MESSAGE from tacking the name of the  
;current routine to the beginning of the message string since  
;!ERR_STRING already contains it.  
MESSAGE, !ERR_STRING, /NONAME, /IOERROR
```

## Obtaining Traceback Information

It is sometimes useful for a procedure or function to obtain information about its caller(s). The `HELP` procedure returns, in a string array, the contents of the procedure stack when the `CALLS` keyword parameter is specified. The first element of the resulting array contains the module name, source filename, and line number of the current level. The second element contains the same information for the caller of the current level, and so on, back to the level of the main program.

For example, the following code fragment prints the name of its caller, followed by the source filename and line number of the call:

```
HELP, CALLS = A

;print 2nd element
PRINT, 'called from:', A[1]
```

This results in a message of the following form:

```
Called from: DIST </usr2/idl/lib/dist.pro (27)>
```

Programs can readily parse the traceback information to extract the source file name and line number.

# Error Handling

IDL contains a system variable that is updated when errors occur. This system variable is described below.

## **!ERROR\_STATE**

This system variable is a structure. Whenever an error occurs, IDL sets the fields in this system variable according to the nature of the field. An IDL error is always comprised of an IDL-generated component, and may also contain an operating system-generated component.

The fields for the `!ERROR_STATE` system variable are described below:

- **NAME** — A read-only string variable containing the error name of the IDL-generated component of the last error message. Although the error code—as defined below in `CODE`—may change between IDL sessions, the name will always remain the same. If an error has not occurred in the current IDL session, this field is set to `IDL_M_SUCCESS`.
- **BLOCK** — A read-only string variable containing the name of the message block for the IDL-generated component of the last error message. If an error has not occurred in the current IDL session, this field is set to `IDL_MBLK_CORE`.
- **CODE** — The error code of the IDL-generated component of the last error in IDL. Whenever an error occurs, IDL sets this system variable to the error code (a negative integer number) of the error. Although the error code may change between IDL sessions, the name—as defined above in `NAME`—will always remain the same. If an error has not occurred in the current IDL session, this field is set to 0.
- **SYS\_CODE** — The error code of the operating system-generated component, if it exists, of the last error. IDL sets this system variable to the OS-defined error code. This field is a two-element longword array. If an error has not occurred in the current IDL session, the array contains all zeros.

On most operating systems, the error code is returned in the first element of the array (i.e., `SYS_CODE[0]`) and the second element is set to 0. Some operating systems (e.g., VMS) can return two separate error codes for some types of filesystem errors. In these cases, `SYS_CODE[1]` is also set to an OS-defined error code.

- **MSG** — The error message of the IDL-generated component of the last error. Whenever an error occurs, IDL sets this field to the error message (a scalar

string) that corresponds to the error code. If an error has not occurred in the current IDL session, this field is set to the null string, ''.

- `SYS_MSG` — The error message of the operating system-generated component, if it exists of the last error. When an operating system error occurs, IDL sets this field to the OS-defined error message string. If an error has not occurred in the current IDL session, this field is set to the null string, ''.
- `MSG_PREFIX` — A string variable containing the prefix string used for the IDL-generated component of error messages.

## Using `!ERROR_STATE`

At the beginning of an IDL session, `!ERROR_STATE` contains default information. To see this information, you can either view `!ERROR_STATE` from the System field of the Variable Watch Window (see [“The Variable Watch Window”](#) on page 619) or you can enter `PRINT, !ERROR_STATE` at the Command Input Line. After an error has occurred, all of the fields of `!ERROR_STATE` display their updated status.

You can use `MESSAGE, /RESET_ERROR STATE` to reset all the fields in `!ERROR_STATE` to their default values.



# Math Errors

The detection of math errors, such as division by zero, overflow, and attempting to take the logarithm of a negative number, is hardware and operating system dependent. Some systems trap more errors than other systems. On systems that implement the IEEE floating-point standard, IDL substitutes the special floating-point values NaN and Infinity when it detects a floating point math error. (See [“Special Floating-Point Values”](#) on page 346.) Integer overflow and underflow is not detected. Integer divide by zero is detected on all platforms.

## A Note on Floating-Point Underflow Errors

Floating-point underflow errors occur when a non-zero result is so close to zero that it cannot be expressed as a normalized floating-point number. In the vast majority of cases, floating-point underflow errors are harmless and can be ignored. For more information on floating-point numbers, see [“Accuracy & Floating-Point Operations”](#) in Chapter 16 of the *Using IDL* manual.

## Accumulated Math Error Status

IDL handles math errors by keeping an accumulated math error status. This status, which is implemented as a longword, contains a bit for each type of math error that is detected by the hardware. When IDL automatically checks and clears this indicator depends on the value of the system variable `!EXCEPT`. The `CHECK_MATH` function also allows you to check and clear the accumulated math error status when desired.

`!EXCEPT` has three possible values:

### **!EXCEPT=0**

Do not report exceptions.

### **!EXCEPT=1**

The default. Report exceptions when the IDL interpreter returns to an interactive prompt. Any math errors that occurred since the last interactive prompt (or call to `CHECK_MATH`) are printed in the IDL command log. A typical message looks like:

```
% Program caused arithmetic error: Floating divide by 0
```

## !EXCEPT=2

Report exceptions after each IDL statement is executed. This setting also allows IDL to report on the program context in which the error occurred, along with the line number in the procedure. A typical message looks like:

```
% Program caused arithmetic error: Floating divide by 0
% Detected at JUNK                3 junk.pro
```

## Special Floating-Point Values

Machines which implement the IEEE standard for binary floating-point arithmetic have two special values for undefined results: NaN (Not A Number) and Infinity. Infinity results when a result is larger than the largest representation. NaN is the result of an undefined computation such as zero divided by zero, taking the square-root of a negative number, or the logarithm of a non-positive number. In many cases, when IDL encounters the value NaN in a data set, it treats it as “missing data.” The special values NaN and Infinity are also accessible in the read-only system variable !VALUES (see “[System Variables](#)” on page 99). These special operands propagate throughout the evaluation process—the result of any term involving these operands is one of these two special values. For example:

```
;Multiply NaN by 3
PRINT, 3 * !VALUES.F_NAN
```

IDL prints:

```
NaN
```

It is important to remember that the value NaN is literally not a number, and as such cannot be compared with a number. For example, suppose you have an array that contains the value NaN:

```
A = [1.0, 2.0, !VALUES.F_NAN]
PRINT, A
```

IDL prints:

```
1.00000      2.00000      NaN
```

If you try to select elements of this array by comparing them with a number (using the [WHERE](#) function, for example), IDL will generate an error:

```
;Print the indices of the elements of A with a value greater than
;one.
PRINT, WHERE(A GT 1.0)
```

IDL prints:

```
1
% Program caused arithmetic error: Floating illegal operand
```

To avoid this problem, use the **FINITE** function to make sure arguments to be compared are in fact valid floating-point numbers:

```
PRINT, WHERE(FINITE(A) EQ 1)
```

IDL prints the indices of the finite elements of A:

```
0          1
```

To then print the indices of the elements of A that are both finite and greater than 1.0, you could use the command:

```
PRINT, WHERE(A[WHERE(FINITE(A) EQ 1)] GT 1.0)
```

IDL prints:

```
1
```

Similarly, if you wanted to find out which elements of an array were *not* valid floating-point numbers, you could use a command like:

```
;Print the indices of the elements of A that are not valid
;floating-point numbers.
PRINT, WHERE(FINITE(A) EQ 0)
```

IDL prints:

```
2
```

Note that the special value Infinity *can* be compared to a floating point number. Thus, if:

```
B = [1.0, 2.0, !VALUES.F_INFINITY]
PRINT, B
```

IDL prints:

```
1.00000      2.00000      Inf
```

and

```
PRINT, WHERE(B GT 1.0)
```

IDL prints:

```
1          2
```

You can also compare numbers directly with the special value Infinity:

```
PRINT, WHERE(B EQ !VALUES.F_INFINITY)
```

IDL prints:

2

## The FINITE Function

Use the FINITE function to explicitly check the validity of floating-point or double-precision operands on machines which use the IEEE floating-point standard. For example, to check the result of the EXP function for validity, use the following statement:

```
;Perform exponentiation.
A = EXP(EXPRESSION)

;Print error message.
IF NOT FINITE(A) THEN PRINT, 'Overflow occurred'
```

If A is an array, use the statement:

```
IF TOTAL(FINITE(A)) NE N_ELEMENTS(A) THEN
```

## Integer Conversions

It must be stressed that when converting from floating to any of the integer types (byte, signed or unsigned short integer, signed or unsigned longword integer, or signed or unsigned 64-bit integer) if overflow is important, you must explicitly check to be sure the operands are in range. Conversions to the above types from floating point, double precision, complex, and string types do not check for overflow—they simply convert the operand to the target integer type, discarding any significant bits of information that do not fit.

When run on a Sun workstation, the program:

```
A = 2.0 ^ 31 + 2
PRINT, LONG(A), LONG(-A), FIX(A), FIX(-A), BYTE(A), BYTE(-A)
```

(which creates a floating-point number 2 larger than the largest positive longword integer), prints the following:

```
2147483647 -2147483648 -1 0 255 0
% Program caused arithmetic error: Floating illegal operand
```

This result is incorrect.

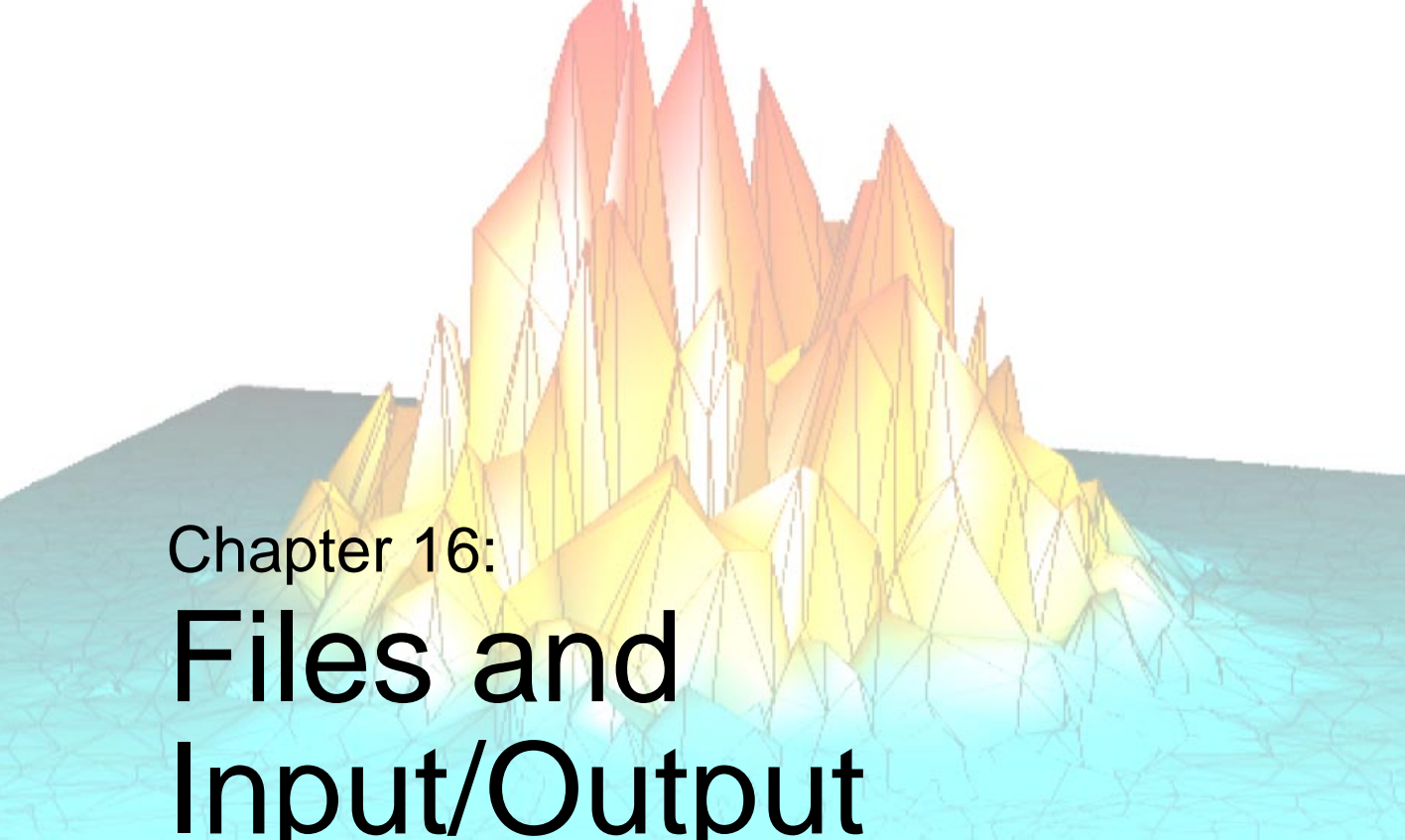
### Warning

No error message will appear if you attempt to convert a floating number whose absolute value is between  $2^{15}$  and  $2^{31} - 1$  to short integer even though the result is

incorrect. Similarly, converting a number in the range of 256 to  $2^{31} - 1$  from floating, complex, or double to byte type produces an incorrect result, but no error message. Furthermore, integer overflow is usually not detected. Your programs must guard explicitly against it.

---





# Chapter 16: Files and Input/Output

The following topics are covered in this chapter:

---

Overview .....	352	Using Unformatted Input/Output .....	394
File I/O in IDL .....	353	Portable Unformatted Input/Output .....	401
Unformatted Input/Output .....	355	Associated Input/Output .....	406
Formatted Input/Output .....	356	File Manipulation Operations .....	411
Opening Files .....	358	UNIX-Specific Information .....	419
Closing Files .....	359	VMS-Specific Information .....	422
Logical Unit Numbers (LUNs) .....	360	Windows-Specific Information .....	432
Reading and Writing Very Large Files ...	363	Macintosh-Specific Information .....	433
Using Free Format Input/Output .....	365	Scientific Data Formats .....	434
Using Explicitly Formatted Input/Output .	370	Support for Standard Image File Formats .	435
Format Codes .....	375		

# Overview

IDL provides powerful facilities for file input and output. Few restrictions are imposed on data files by IDL, and there is no unique IDL format. This chapter describes IDL input/output methods and routines and gives examples of programs which read and write data using IDL, C, and FORTRAN.

The first section of this chapter provides a description for how IDL input/output works. It is intentionally brief and is intended to serve only as an introduction. Additional details are covered in the following sections. For the IDL user, perhaps the largest single difference between platforms is input/output. The majority of this chapter covers information that is required in all of the environments IDL supports. Operating system specific information is concentrated in the final sections of this chapter.



## File I/O in IDL

Before any file input or output can be performed, it is necessary to open a file. This is done using either the OPENR (Open for Reading), OPENW (Open for Writing), or OPENU (Open for Update) procedures. When a file is opened, it is associated with a *Logical Unit Number*, or LUN. All file input and output routines in IDL use the LUN rather than the filename, and most require that the LUN be explicitly specified. Once a file is opened, several input/output routines are available for use. Each routine fills a particular need – the one to use depends on the particular situation.

There are three exceptions to the need to open any file before performing input/output on it. Three files are always open – in fact, the user is not allowed to close them. These files are the *standard input* (usually the keyboard), the *standard output* (usually the IDL log window), and the *standard error output* (usually the terminal screen). These three files are associated with LUNs 0, -1, and -2, respectively. Because these files are always open, there is no need to open them prior to using them for input/output. The READ and PRINT procedures automatically use these files, so basic formatted input/output is extremely simple.

### Simple Examples

It is easy to use input/output using the default input and output files. The IDL command:

```
PRINT, 'Hello World.'
```

causes IDL to print the line:

```
Hello World.
```

on the terminal screen. This happens because PRINT formats its arguments and prints them to LUN -1, which is the standard output file. It is only slightly more complicated to use other files. The following IDL statements show how the above “Hello World” example could be sent to a file named *hello.dat*:

```
;Open LUN 1 for hello.dat with write access.  
OPENW, 1, 'hello.dat'  
  
;Do the output operation to the file.  
PRINTF, 1, 'Hello World.'  
  
;Close the file.  
CLOSE, 1
```

## Routines for Input/Output

The following routines are useful when doing input/output operations. For more information on these commands, see *IDL Reference Guide*.

Routine	Description
<a href="#">ASSOC</a>	Map an array structure to a data file. ASSOC provides extremely efficient direct access to unformatted data and is an important IDL function to understand.
<a href="#">EOF</a>	Check for the end-of-file condition.
<a href="#">FINDFILE</a>	Locate files that match a file specification.
<a href="#">FLUSH</a>	Ensure all buffered data for a LUN has actually been output to the file.
<a href="#">FREE_LUN</a>	Free unique file units.
<a href="#">FSTAT</a>	Get detailed information about any LUN.
<a href="#">GET_KBRD</a>	Read single characters from the keyboard.
<a href="#">GET_LUN</a>	Allocate and free unique file units.
<a href="#">HELP, /FILES</a>	Print information about open files.
<a href="#">PNT_LINE</a>	Position the file pointer.
<a href="#">PRINT/PRINTF</a>	<ul style="list-style-type: none"> <li>• PRINT outputs formatted data to the standard output file (LUN -1).</li> <li>• PRINTF outputs formatted data to the specified LUN.</li> </ul>
<a href="#">READ/READF READS READU</a>	<ul style="list-style-type: none"> <li>• READ reads formatted data from the standard input file (LUN 0).</li> <li>• READF reads formatted data from the specified LUN.</li> <li>• READS performs formatted input from a string variable.</li> <li>• READU Input unformatted data from the specified LUN.</li> </ul>
<a href="#">WRITEU</a>	Output unformatted data to the specified LUN.

Table 16-1: Routines for Input/Output

# Unformatted Input/Output

Unformatted Input/Output is the most basic form of input/output. Unformatted input/output transfers the internal binary representation of the data directly between memory and the file.

## Advantages of Unformatted I/O

Unformatted input/output is the simplest and most efficient form of input/output. It is usually the most compact way to store data.

## Disadvantages of Unformatted I/O

Unformatted input/output is the least portable form of input/output. Unformatted data files can only be moved easily to and from computers that share the same internal data representation. It should be noted that XDR (eXternal Data Representation) files, described in [“Portable Unformatted Input/Output”](#) on page 401, can be used to produce portable binary data.

Unformatted input/output is not directly human readable, so you cannot type it out on a terminal screen or edit it with a text editor.

# Formatted Input/Output

Formatted output converts the internal binary representation of the data to ASCII characters which are written to the output file. Formatted input reads characters from the input file and converts them to internal form. Formatted I/O can be either “Free” format or “Explicit” format, as described below.

## Advantages of Formatted I/O

Formatted input/output is very portable. It is a simple process to move formatted data files to various computers, even computers running different operating systems, as long as they all use the ASCII character set. (ASCII is the American Standard Code for Information Interchange. It is the character set used by almost all current computers, with the notable exception of large IBM mainframes.)

Formatted files are human readable and can be typed to the terminal screen or edited with a text editor.

## Disadvantages of Formatted I/O

Formatted input/output is more computationally expensive than unformatted input/output because of the need to convert between internal binary data and ASCII text. Formatted data requires more space than unformatted to represent the same information. Inaccuracies can result when converting data between text and the internal representation.

## Free Format I/O

With free format input/output, IDL uses default rules to format the data.

### Advantages of Free Format I/O

The user is free of the chore of deciding how the data should be formatted. Free format is extremely simple and easy to use. It provides the ability to handle the majority of formatted input/output needs with a minimum of effort.

### Disadvantages of Free Format I/O

The default formats used are not always exactly what is required. In this case, explicit formatting is necessary.

## Explicit Format I/O

Explicit format I/O allows you to specify the exact format for input/output.

## Advantages of Explicit I/O

Explicit formatting allows a great deal of flexibility in specifying exactly how data will be formatted. Formats are specified using a syntax that is similar to that used in FORTRAN format statements. Scientists and engineers already familiar with FORTRAN will find IDL formats easy to write. Commonly used FORTRAN format codes are supported. In addition, IDL formats have been extended to provide many of the capabilities found in the *scanf()* and *printf()* functions commonly found in the C language runtime library.

## Disadvantages of Explicit I/O

Using explicitly specified formats requires the user to specify more detail—they are, therefore, more complicated to use than free format.

The type of input/output to use in a given situation is usually determined by considering the advantages and disadvantages of each method as they relate to the problem to be solved. Also, when transferring data to or from other programs or systems, the type of input/output is determined by the application. The following suggestions are intended to give a rough idea of the issues involved, though there are always exceptions:

- Images and large data sets are usually stored and manipulated using unformatted input/output in order to minimize processing overhead. The IDL ASSOC function is often the natural way to access such data.
- Data that need to be human readable should be written using formatted input/output.
- Data that need to be portable should be written using formatted input/output. Another option is to use unformatted XDR files by specifying the XDR keyword with the OPEN procedures. This is especially important if moving between computers with markedly different internal binary data formats. XDR is discussed in “[Portable Unformatted Input/Output](#)” on page 401.
- Free format input/output is easier to use than explicitly formatted input/output and about as easy as unformatted input/output, so it is often a good choice for small files where there is no strong reason to prefer one method over another.

## Opening Files

Before a file can be processed by IDL, it must be opened using one of the procedures described in the following table. All open files are associated with a LUN (Logical Unit Number) within IDL, and all input/output routines refer to files via this number. For example, to open the file named *data.dat* for reading on file unit 1, use the following statement:

```
OPENR, 1, 'data.dat'
```

The OPEN procedures can be used with certain keywords to modify their normal behavior. Some keywords are generally applicable, while others only have effect under a given operating system. Some operating system specific keywords are allowed (and ignored) under other operating systems in order to facilitate writing portable routines.

Procedure	Description
OPENR	Opens an existing file for input only.
OPENW	Opens a new file for input and output. Under UNIX, Windows, and on the Macintosh, if the named file already exists, its old contents are overwritten. Under VMS, a file with the same name and a higher version number is created.
OPENU	Opens an existing file for input and output.

*Table 16-2: IDL File Opening Commands*

### Platform-Specific Keywords to the OPEN Procedure

Different computers and operating systems perform input/output in different ways. See [OPEN](#) in the *IDL Reference Guide* for keywords to the OPEN procedures that apply under UNIX, VMS, Windows, or the Macintosh OS.

# Closing Files

After work involving the file is complete, it should be closed. Closing a file removes the association between the file and its unit number, thus freeing the unit number for use with a different file. There is usually an operating system-imposed limit on the number of files a user may have open at once. Although this number is large enough that it rarely causes problems, situations can occur where a file must be closed before another file may be opened. In any event, it is good style to only keep needed files open.

There are three ways to close a file:

- Use the `CLOSE` procedure.
- Use the `FREE_LUN` procedure on a LUN that has been allocated by `GET_LUN`.
- Exit IDL. IDL closes all open files when it exits.

Calling the `CLOSE` procedure is the most common way to close a file unit. For example, to close file unit number 1, use the following statement:

```
CLOSE, 1
```

In addition, if `FREE_LUN` is called with a file unit number that was previously allocated by `GET_LUN`, it calls `CLOSE` before deallocating the file unit. Finally, all open files are automatically closed when IDL exits.

# Logical Unit Numbers (LUNs)

IDL Logical Unit Numbers (LUNs) fall within the range -2 to 128. Some LUNs are reserved for special functions as described below.

## The Standard Input, Output, and Error LUNs

The three LUNs described below have special meanings that are operating system dependent:

### UNIX

Logical Unit Numbers 0, -1, and -2 are tied to *stdin*, *stdout*, and *stderr*, respectively. This means that the normal UNIX file redirection and pipe operations work with IDL. For example, the shell command

```
%idl < idl.inp >& idl.out &
```

will cause IDL to execute in the background, reading its input from the file *idl.inp* and writing its output to the file *idl.out*. Any messages sent to *stderr* are also sent to *idl.out*.

When using the IDL Development Environment (IDLDE), Logical Unit Numbers 0, -1, and -2 are tied to *stdin* (the command line), *stdout* (the log window), and *stderr* (the log window), respectively.

### VMS

Logical Unit Numbers 0, -1, and -2 are tied to *SY\$INPUT*, *SY\$OUTPUT*, and *SY\$ERROR* respectively. This means that the DCL *DEFINE* command can be used to redefine where IDL gets commands and writes its output. It also means that IDL can be used in command and batch files.

When using the IDL Development Environment (IDLDE), Logical Unit Numbers 0, -1, and -2 are tied to *SY\$INPUT* (the command line), *SY\$OUTPUT* (the log window), and *SY\$ERROR* (the log window), respectively.

### Windows and Macintosh

Logical Unit Numbers 0, -1, and -2 are tied to *stdin* (the command line), *stdout* (the log window), and *stderr* (the log window), respectively.

These special file units are described in more detail below.



## File Unit 0

This LUN represents the standard input stream, which is usually the keyboard. Therefore, the IDL statement:

```
READ, X
```

is equivalent to the following:

```
READF, 0, X
```

## File Unit -1

This LUN represents the standard output stream, which is usually the terminal screen. Therefore, the IDL statement:

```
PRINT, X
```

is equivalent to the following:

```
PRINTF, -1, X
```

## File Unit -2

This LUN represents the standard error stream, which is usually the terminal screen.

## File Units 1–99

These are the file units for normal interactive use. When using IDL interactively, the user arbitrarily selects the file units used. The file units from 1 to 99 are available for this use.

## File Units 100–128

These are the file units managed by the `GET_LUN` and `FREE_LUN` procedures. If an IDL procedure or function that uses files is written to explicitly use a given file unit, there is a chance that it will conflict with other routines that use the same unit. It is therefore necessary to avoid explicit file unit numbers when writing IDL procedures and functions. The `GET_LUN` and `FREE_LUN` procedures provide a standard mechanism for IDL routines to obtain unique file units. `GET_LUN` allocates a file unit from a pool of free units in the range 100 to 128. This unit will not be allocated again until it is released by a call to `FREE_LUN`. Meanwhile, it is available for the exclusive use of the program that allocated it. A typical procedure that needs a file unit might be structured as follows:

```
PRO DEMO
;Get a unique file unit and open the file.
OPENR, UNIT, /GET_LUN
```

```
;Body of program goes here.  
.  
.  
.  
  
;Return file unit.  
FREE_LUN, UNIT  
  
;Since the file is still open, FREE_LUN will automatically call  
;CLOSE.  
END
```

**Note**

---

All IDL procedures and functions that open files should use GET\_LUN/ FREE\_LUN to obtain file units. Furthermore, the file units between 100 and 128 should never be used unless previously allocated by GET\_LUN.

---

## Reading and Writing Very Large Files

IDL on all platforms is able to read and write data from files up to  $2^{31}-1$  bytes in length. On some platforms, it is also able to read and write data from files longer than this limit. Currently, IDL on Digital Unix, HP-UX, Sun Solaris (SPARC and x86), and SGI IRIX is able to perform I/O on such files. RSI expects this list of platforms to increase with future releases.

When reading and writing to files smaller than this limit, there is no difference in behavior between the platforms that can and those that cannot handle larger files. IDL uses longword integers for file position arguments (e.g. POINT\_LUN, FSTAT) and keywords, as before. However, when dealing with files that exceed this limit, IDL uses signed 64-bit integers in order to be able to properly represent the offset. Consider the following example:

```
;Open the file
OPENW, 1, 'test.dat'

;Initial position should be 0.
POINT_LUN, -1, POS

;Print the position and its type.
HELP, POS

;Move the file pointer past the signed 32-bit boundary.
POINT_LUN, 1, '000000fffffffffff'x

;The position is now too large to represent as a longword.
POINT_LUN, -1, POS

;Print the position and its type.
HELP, POS

CLOSE, 1
```

Executing these statements results in the following output:

```
POS          LONG          =          0
POS          LONG64       =         1099511627775
```

Initially, the file position is 0, which fits easily into a 32-bit integer. Once the file position exceeds the range of a signed 32-bit number, IDL automatically shifts to the 64-bit integer type.

## Limitations of Large File Support

There are limitations on IDL's support for very large files that must be understood by the IDL programmer:

- On any platform, the amount of data that IDL can transfer in a single operation is limited by the amount of memory it can allocate. Currently, IDL is a 32-bit program, and as such, can theoretically address up to  $2^{31}-1$  bytes of memory (approximately 2.3GB). Reading or writing data larger than this limit must be done in multiple operations. Most systems do not have 2.3 GB of memory available, and other programs running on the system also compete for the same memory, so the actual memory available is likely to be considerably smaller.
- The ability to read or write to very large files is constrained by the ability of the underlying file system to support such files. Many platforms can only support large files on certain file systems. For example, many platforms will be unable to support these operations on NFS mounted file systems because the latest version of NFS must be in use on both client and server. Some platforms, such HP-UX, can only support such operations on special large file systems, and only if they are mounted using the appropriate mount options. Consult your system documentation to determine the limitations present on your system and the procedures for supporting very large file.

# Using Free Format Input/Output

Use of formatted data is most appropriate when the data must be in human readable form, such as when it is to be prepared or modified with a text editor. Formatted data also are highly portable between various computers and operating systems.

In addition to the PRINT, PRINTF, READ, and READF routines already discussed, the STRING function can be used to generate formatted output that is sent to a string variable instead of a file. The READS procedure can be used to read formatted input from a string variable.

The exact format of the character data may be specified to these routines by providing a format string via the FORMAT keyword. If no format string is given, default formats for each type of data are applied. This method of formatted input/output is called free format. Free format input/output is suitable for most applications involving formatted data. It is designed to provide input/output abilities with a minimum of programming.

## Structures and Free Format Input/Output

IDL structures present a special problem for default formatted input and output. The default format for displaying structure data is to surround the structure with curly braces ({}). For example, if you define an anonymous structure:

```
struct = { A:2, B:3, C:'A String' }
```

and then use default formatted output via the PRINT command:

```
PRINT, struct
```

IDL prints:

```
{          2          3 A String}
```

You might suppose that default formatted input would recognize that the curly braces are part of the formatting and ignore them. This is not the case, however. By default, to read the third field in the structure (the string field) IDL will read from the “A” to the end of the line, including the closing brace.

This behavior, while unsymmetric, seems to be the best choice for default behavior—displaying the result of the PRINT statement on the computer screen. We recommend that you use explicitly formatted input/output when reading and writing structures to disk files, so as not to have to explicitly code around the possibility that your structure may include strings.

## Free Format Input

The following rules are used by IDL to perform free format input:

1. Input is performed on scalar variables. Array and structure variables are treated as collections of scalar variables. For example,

```
A = INTARR(5)
READ, A
```

causes IDL to read five separate values to fill each element of the variable A.

2. If the current input line is empty and there are variables left requiring input, read another line.
3. If the current input line is not empty but there are no variables left requiring input, the remainder of the line is ignored.
4. Input data must be separated by commas or white space (tabs, spaces, or new lines).
5. When reading into a variable of type string, all characters remaining in the current input line are placed into the string.
6. When reading into numeric variables, every effort is made to convert the input into a value of the expected type. Decimal points are optional and exponential (scientific) notation is allowed. If a floating-point datum is provided for an integer variable, the value is truncated.
7. When reading into a variable of complex type, the real and imaginary parts are separated by a comma and surrounded by parentheses. If only a single value is provided, it is taken as the real part of the variable, and the imaginary part is set to zero. For example:

```
;Create a complex variable.
A = COMPLEX(0)

;IDL prompts for input with a colon:
READ, A

;The user enters "(3,4)" and A is set to COMPLEX(3, 4).
:(3, 4)

;IDL prompts for input with a colon:
READ, A

;The user enters "50" and A is set to COMPLEX(50, 0).
:50
```

## Free Format Output

The following rules are used by IDL to perform free format output:

1. The format used to output numeric data is determined by the data type. The formats used are summarized in the table below. The formats are specified in the FORTRAN-like style used by IDL for explicitly formatted input/output.

Data Type	Format
Byte	I4
Int, UInt	I8
Long, ULong	I12
Float	G13.6
Long64, ULong64	I22
Double	G16.8
Complex	'(, G13.6, ',', G13.6, )'
Double-precision Complex	'(, G16.8, ',', G16.8, )'
String	Output full string on current line.

*Table 16-3: Formats Used for Free-Format Output*

2. The current output line is filled with characters until one of the following happens:
  - A. There is no more data to output.
  - B. The output line is full. When output is to a file, the default line width is 80 columns (you can override this default by setting the WIDTH keyword to the OPEN procedure). When the output is to the standard output, IDL uses the current width of your tty or command log window.
  - C. An entire row is output in the case of multidimensional arrays.
3. When outputting a structure variable, its contents are bracketed with “{” and “}” characters.

## Example: Free Format Input/Output

IDL free format input/output is extremely easy to use. The following IDL statements demonstrate how to read into a complicated structure variable and then print the results:

```
;Create a structure named "types" that contains seven of the basic
;IDL data types, as well as a floating-point array.
A = {TYPES, A:0B, B:0, C:0L, D:1.0, E:1D, $
     F:COMPLEX(0), G: 'string', E:FLTARR(5)}

;Read free-formatted data from input
READ, A

;IDL prompts for input with a colon. We enter values for the first
;six numeric fields of A and the string.
: 1 2 3 4 5 (6,7) EIGHT
```

Notice that the complex value was specified as (6, 7). If the parentheses had been omitted, the complex field of A would have received the value COMPLEX(6, 0), and the 7 would have been input for the next field. When reading into a string variable, IDL starts from the current point in the input and continues to the end of the line. Thus, we do not enter values intended for the rest of the structure on this line.

```
;There are still fields of A that have not received data, so IDL
;prompts for another line of input.
: 9 10 11 12 13

;Show the result.
PRINT, A
```

Executing these statements results in the following output:

```
{ 1          2          3          4.00000          5.00000000
  (    6.00000,    7.00000) eight
    9.00000    10.0000    11.0000    12.0000    13.0000
  }
```

When producing the output, IDL uses default rules for formatting the values and attempts to place as many items as possible onto each line. Because the variable A is a structure, braces {} are placed around the output. As noted above, when IDL reads strings it continues to the end of the line. For this reason, it is usually convenient to place string variables at the end of the list of variables to be input. For example, if S is a string variable and I is an integer:

```
;Read into the string first.
READ, S, I
```



```
;IDL prompts for input. We enter a string value followed by an  
;integer.  
: Hello World 34  
  
;The entire previous line was placed into the string variable S,  
;and I still requires input. IDL prompts for another line.  
: 34
```

# Using Explicitly Formatted Input/Output

The `FORMAT` keyword can be used with the formatted input/output routines to explicitly specify the appearance of the data. The syntax of IDL format strings is extremely similar to that used in FORTRAN. The format string specifies the format in which data is to be transferred as well as the data conversion required to achieve that format. The format specification strings supplied by the `FORMAT` keyword have the form:

```
FORMAT = '(q1f1s1f2s2 ... fnqn)'
```

where `q`, `f`, and `s` are described below.

## Record Terminators

`q` is zero or more slash (/) record terminators. On output, each record terminator causes the output to move to a new line. On input, each record terminator causes the next line of input to be read.

## Format Codes

`f` is a format code. Some format codes specify how data should be transferred while others control some other function related to how input/output is handled. The code `f` can also be a nested format specification enclosed in parentheses. This is called a *group specification* and has the following form:

```
...[n](q1f1s1f2s2 ... fnqn) ...
```

A group specification consists of an optional repeat count `n` followed by a format specification enclosed in parentheses. Use of group specifications allows more compact format specifications to be written. For example, the format specification:

```
FORMAT = '("Result: ", "<", I5, ">", "<", I5, ">")'
```

can be written more concisely using a group specification:

```
FORMAT = '("Result: ", 2("<", I5, ">"))'
```

If the repeat count is 1 or is not given, the parentheses serve only to group format codes for use in format reversion (discussed in the next section).

## Field Separators

`s` is a field separator. A field separator consists of one or more commas (,) and/or slash record terminators (/). The only restriction is that two commas cannot occur side-by-side.

The arguments provided in a call to a formatted input/output routine are called the *argument list*. The argument list specifies the data to be moved between memory and the file. All data are handled in terms of basic IDL components. Thus, an array is considered to be a collection of scalar data elements, and a structure is processed in terms of its basic components. Complex scalar values are treated as two floating-point values.

## Rules for Explicitly Formatted Input/Output

IDL uses the following rules to process explicitly formatted input/output:

1. Traverse the format string from left to right, processing each record terminator and format code until an error occurs or no data is left in the argument list. The comma field separator serves no purpose except to delimit the format codes.
2. It is an error to specify an argument list with a format string that does not contain a format code that transfers data to or from the argument list because an infinite loop would result.
3. When a slash record terminator (/) is encountered, the current record is completed, and a new one is started. For output, this means that a new line is started. For input, it means that the rest of the current input record is ignored, and the next input record is read.
4. When a format code that does not transfer data to or from the argument list is encountered, process it according to its meaning. The format codes that do not

transfer data to or from the argument list are summarized in the following table:

<b>Code</b>	<b>Action</b>
Quoted String	On output, the contents of the string are written out. On input, quoted strings are ignored.
:	The colon format code in a format string terminates format processing if no more items remain in the argument list. It has no effect if data still remains on the list.
\$	On output, if a \$ format code is placed anywhere in the format string, the new line implied by the closing parenthesis of the format string is suppressed. On input, the \$ format code is ignored.
nH	FORTRAN-style Hollerith string. Hollerith strings are treated exactly like quoted strings.
nX	Skips $n$ character positions.
Tn	Tab. Sets the character position of the next item in the current record.
TLn	Tab Left. Specifies that the next character to be transferred to or from the current record is the $n$ -th character to the left of the current position.
TRn	Tab Right. Specifies that the next character to be transferred to or from the current record is the $n$ -th character to the right of the current position.

*Table 16-4: Format Codes that do not Transfer Data*

5. When a format code that transfers data to or from the argument list is encountered, it is matched up with the next datum in the argument list. The

format codes that transfer data to or from the argument list are summarized in the following table:

<b>Code</b>	<b>Action</b>
A	Transfer character data.
C()	Transfer calendar (Julian date and/or time) data.
D	Transfer double-precision, floating-point data.
E	Transfer floating-point data using scientific (exponential) notation.
F	Transfer floating-point data.
G	Use F or E format depending on the magnitude of the value being processed.
I	Transfer integer data.
O	Transfer octal data.
Q	Obtain the number of characters in the input record remaining to be transferred during a read operation. In an output statement, the Q format code has no effect except that the corresponding input/output list element is skipped.
Z	Transfer Hexadecimal data.

*Table 16-5: Format Codes that Transfer Data*

6. On input, read data from the file and format it according to the format code. If the data type of the input data does not agree with the data type of the variable that is to receive the result, do the type conversion if possible; otherwise, issue a type conversion error and stop.
7. On output, write the data according to the format code. If the data type does not agree with the format code, do the type conversion prior to doing the output if possible. If the type conversion is not possible, issue a type conversion error and stop.
8. If the last closing parenthesis of the format string is reached and there are no data left on the argument list, then format processing terminates. If, however, there are still data to be processed on the argument list, then part or all of the format specification is reused. This process is called format reversion.

## Format Reversion

In format reversion, the current record is terminated, a new one is initiated, and format control reverts to the group repeat specification whose opening parenthesis matches the next-to-last closing parenthesis of the format string. If the format does not contain a group repeat specification, format control returns to the initial opening parenthesis of the format string. For example, the IDL command:

```
PRINT, FORMAT = '("The values are: ", 2("<", I1, ">"))', $
      INDGEN(6)
```

results in the output

```
The values are: <0><1>
<2><3>
<4><5>
```

The process involved in generating this output is as follows:

1. Output the string “The values are: ”.
2. Process the group specification and output the first two values. The end of the format specification is encountered, so end the output record. Data are remaining, so move back to the group specification
 

```
      2("<", I1, ">")
```

 by format reversion.
3. Repeat Step 2 until no data remain. End the output record. Format processing is complete.

# Format Codes

## “A” Format Code

The A format code transfers character data. The format is

```
[n]A[w]
```

where:

$n$  — is an optional repeat count ( $1 \leq n \leq 32767$ ) specifying the number of times the format code should be processed. If  $n$  is not specified, a repeat count of one is used.

$w$  — is an optional width ( $1 \leq w \leq 256$ ) specifying the number of characters to be transferred. If  $w$  is not specified, the entire string is transferred. On output, if  $w$  is greater than the length of the string, the string is right justified. On input, IDL strings have dynamic length, so  $w$  specifies the resulting length of input string variables.

For example, the IDL statement,

```
PRINT, FORMAT = '(A6)', '123456789'
```

generates the following output:

```
123456
```

### Note

---

While an IDL string variable can hold up to 64 Kbytes of information, the buffer that handles input at the IDL command prompt is limited to 255 characters. If for some reason you need to create a string variable longer than 255 characters at the IDL command prompt, split the variable into multiple sub-variables and combine them with the “+” operator:

```
var = var1+var2+var3
```

This limit only affects string constants created at the IDL command prompt.

---

## “:” Format Code

The colon format code terminates format processing if there are no more data remaining in the argument list. For example, the IDL statement,

```
PRINT, FORMAT = '(6(I1, :, ", " ))', INDGEN(6)
```

will output the following comma-separated list of integer values:

0, 1, 2, 3, 4, 5

The use of the colon format code prevented a comma from being output following the final item in the argument list.

## “\$” Format Code

When IDL completes output format processing, it normally outputs a newline to terminate the output operation. However, if a “\$” format code is found in the format specification, this default newline is not output. The “\$” format code is only used on output; it is ignored during input formatting. The most common use for the “\$” format code is in prompting for user input. For example, the IDL statements,

```
;Prompt for input. Suppress the carriage return.
PRINT, FORMAT = '($, "Enter value: ")'

;Read the response.
READ, VALUE
```

will prompt for input without forcing the user’s response to appear on a separate line from the prompt. Under VMS, the “\$” format code does not work with files opened with carriage-return carriage control, which is the default for new files. However, it does work with explicit or FORTRAN carriage control. FORTRAN carriage control is described in [“Reading FORTRAN-Generated Unformatted Data with IDL”](#) on page 419.

## “F,” “D,” “E,” and “G” Format Codes

The F, D, E, and G format codes are used to transfer floating-point values between memory and the specified file. The format is

```
[n]F[w.d]
[n]D[w.d]
[n]E[w.d] or [n]E[w.dEe]
[n]G[w.d] or [n]G[w.dEe]
```

where

$n$  — is an optional repeat count ( $1 \leq n \leq 32767$ ) specifying the number of times the format code should be processed. If  $n$  is not specified, a repeat count of 1 is used.

$w.d$  — is an optional width specification ( $1 \leq w \leq 256$ ,  $1 \leq d < w$ ). The variable  $w$  specifies the number of characters in the external field. For the F, D, and E format codes,  $d$  specifies the number of positions after the decimal point. For the G format code,  $d$  specifies the number of significant digits displayed.



$e$  — is an optional width ( $1 \leq e \leq 256$ ) specifying the width of exponent part of the field. IDL ignores this value—it is allowed for compatibility with FORTRAN.

On input, the F, D, E, and G format codes all transfer  $w$  characters from the external field and assign them as a real value to the corresponding input/output argument list datum.

The F and D format codes are used to output values using fixed-point notation. The value is rounded to  $d$  decimal positions and right-justified into an external field that is  $w$  characters wide. The value of  $w$  must be large enough to include a minus sign when necessary, at least one digit to the left of the decimal point, the decimal point, and  $d$  digits to the right of the decimal point. The code D is identical to F (except for its default values for  $w$  and  $d$ ) and exists in IDL primarily for compatibility with FORTRAN.

The E format code is used for scientific (exponential) notation. The value is rounded to  $d$  decimal positions and right-justified into an external field that is  $w$  characters wide. The value of  $w$  must be large enough to include a minus sign when necessary, at least one digit to the left of the decimal point, the decimal point,  $d$  digits to the right of the decimal point, a plus or minus sign for the exponent, the character “e” or “E”, and at least two characters for the exponent.

---

**Note**

IDL uses a standard I/O function to format numbers and their exponents. As a result, different platforms may print different numbers of exponent digits.

---

The G format code uses the F output style when reasonable and E for other values, but displays exactly  $d$  significant digits rather than  $d$  digits following the decimal point.

On output, if the field provided is not wide enough, it is filled with asterisks (\*) to indicate the overflow condition. If  $w$  is zero, the “natural” width for the value is used—the value is read or output using a default format without any leading or

trailing whitespace, in the style of the C standard input/output library *printf* (3S) function. If *w*, *d*, or *e* are omitted, the values specified in the following table are used.

<b>Data Type</b>	<b><i>w</i></b>	<b><i>d</i></b>	<b><i>e</i></b>
Float, Complex	15	7	2 (3 for Windows)
Double	25	16	2 (3 for Windows)
All Other Types	25	16	2 (3 for Windows)

*Table 16-6: Floating Format Defaults*

Using a value of zero for the *w* parameter is useful when reading tables of data in which individual elements may be of varying lengths. For example, if your data reside in tables of the following form:

```
26.01 92.555 344.2
101.0 6.123 99.845
23.723 200.02 141.93
```

setting the format to

```
FORMAT = '(3F0)'
```

ensures that the correct number of digits are read or output for each element.

Normally, the case of the format code is ignored by IDL. However, the case of the E and G format codes determines the case used to output the exponent in scientific notation. The following table gives examples of several floating-point formats and the resulting output.

<b>Format</b>	<b>Internal Value</b>	<b>Formatted Output</b>
F	100.0	<i>bbbb</i> 100.0000000
F	100.0D	<i>bbbbbb</i> 100.000000000000000000
F10.0	100.0	<i>bbbbbb</i> 100.
F10.1	100.0	<i>bbbbbb</i> 100.0
F10.4	100.0	100.0000
F2.1	100.0	**

*Table 16-7: Floating-Point Output Examples (“b” represents a blank space)*

Format	Internal Value	Formatted Output
e10.4	100.0	1.0000e+02 (e+03 for Windows)
E10.4	100.0	1.0000E+02 (e+03 for Windows)
g10.4	100.0	100.0
g10.4	10000000.0	1.000e+07

Table 16-7: Floating-Point Output Examples (“b” represents a blank space)

## “I,” “O,” and “Z” Format Codes

The I, O, and Z format codes are used to transfer integer values to and from the specified file. The I format code is used to output decimal values, O is used for octal values, and Z is used for hexadecimal values.

The format is as follows:

```
[n]I[w] or [n]I[w.m]
[n]O[w] or [n]O[w.m]
[n]Z[w] or [n]Z[w.m]
```

where

*n* — is an optional repeat count ( $1 \leq n \leq 32767$ ) specifying the number of times the format code should be processed. If *n* is not specified, a repeat count of 1 is used.

*w* — is an optional integer value ( $1 \leq w \leq 256$ ) specifying the width of the field in characters. The default values used if *w* is omitted are specified in the following table:

Data Type	<i>w</i>
Byte, Int, UInt	7
Long, ULong, Float	12
Long64, ULong64	22
Double	23
All Other Types	12

Table 16-8: Integer Format Defaults

If the field provided is not wide enough, it is filled with asterisks (\*) to indicate the overflow condition. If  $w$  is zero, the “natural” width for the value is used—the value is read or output using a default format without any leading or trailing white space, in the style of the C standard input/output library *printf* (3S) function.

Note that using a value of zero for the  $w$  parameter is useful when reading tables of data in which individual elements may be of varying lengths. For example, if your data reside in tables of the following form:

```
26 92 344
101 6 99
23 200 141
```

setting the format to

```
FORMAT = '(3I0)'
```

ensures that the correct number of digits are read or output for each element.

$m$  — On output,  $m$  specifies the minimum number of nonblank digits required ( $1 \leq m \leq 256$ ). The field is zero-filled on the left if necessary. If  $m$  is omitted or zero, the external field is blank filled.

Normally, the case of the format code is ignored by IDL. However, the case of the Z format codes determines the case used to output the hexadecimal digits A-F. The following table gives examples of several integer formats and the resulting output.

Format	Internal Value	Formatted Output
I	3000	bbb3000
I6.5	3000	b03000
I5.6	3000	*****
I2	3000	**
O	3000	bbb5670
O6.5	3000	b05670
O5.6	3000	*****
O2	3000	**
z	3000	bbbbbb8

Table 16-9: Integer Output Examples (“b” represents a blank space)

Format	Internal Value	Formatted Output
Z	3000	bbbbBB8
Z6.5	3000	b00bb8
Z5.6	3000	*****
Z2	3000	**

Table 16-9: Integer Output Examples (“b” represents a blank space)

## “Q” Format Code

The Q format code returns the number of characters in the input record remaining to be transferred during the current read operation. It is ignored during output formatting. Format Q is useful for determining how many characters have been read on a line. For example, the following IDL statements count the number of characters in file *demo.dat*:

```

;Open file for reading.
OPENR, 1, "demo.dat"

;Create a longword integer to keep the count.
N = 0L

;Count the characters.
WHILE(NOT EOF(1)) DO BEGIN
    READF, 1, CUR, FORMAT = '(q)' & N = N + CUR
END

;Report the result.
PRINT, FORMAT = '("counted", N, "characters.")'

;Close file.
CLOSE, 1

```

## Quoted String and “H” Format Codes

On output, any quoted strings or Hollerith constants are sent directly to the output. On input, they are ignored. For example, the IDL statement,

```
PRINT, FORMAT = '("Value: ", I0)', 23
```

results in the following output:

```
Value: 23
```

Notice the use of single quotes around the entire format string and double quotes around the quoted string inside the format. This is necessary because we are including quotes inside a quoted string. It would have been equally correct to use double quotes around the entire format string and single quotes internally. Another way to specify the string is with a Hollerith constant as follows:

```
PRINT, FORMAT = '(7HValue: , I0)', 23
```

The format for a Hollerith constant is:

$$nHc_1c_2c_3 \dots c_n$$

where

$n$  — is the number of characters in the constant ( $1 \leq n \leq 255$ ).

$c_i$  — is the characters that make up the constant. The number of characters must agree with the value provided for  $n$ .

## “T” Format Code

The T format code specifies the absolute position in the current record. The format is

$$Tn$$

where

$n$  — is the absolute character position within the record to which the current position should be set ( $1 \leq n \leq 32767$ ).

T — differs from the TL, TR, and X format codes primarily in that it requires an absolute position rather than an offset from the current position. For example,

```
PRINT, FORMAT = '("First", 20X, "Last", T10, "Middle")'
```

produces the following output:

```
FirstbbbbMiddlebbbbbbbbLast
```

where “*b*” represents a blank space.

## “TL” Format Code

The TL format code moves the current position in the external record to the left. The format is

$$TLn$$

where

$n$  — is the number of characters to move left from the current position ( $1 \leq n \leq 32767$ ). If the value of  $n$  is greater than the current position, the current position is moved to column one.

TL — is used to move backwards in the current record. It can be used on input to read the same data twice or on output to position the output nonsequentially. For example,

```
PRINT, FORMAT = '("First", 20X, "Last", TL15, "Middle")'
```

produces the following output:

```
FirstbbbbbbbbMiddlebbbbLast
```

where “*b*” represents a blank space.

## “TR” and “X” Format Codes

The TR and X format codes move the current position in the record to the right. The format is

```
TRn
nX
```

where

$n$  — is the number of characters to skip ( $1 \leq n \leq 32767$ ). On input,  $n$  characters in the current input record will be passed over. On output, the current output position is moved  $n$  characters to the right.

The TR or X format codes can be used to leave whitespace in the output or to skip over unwanted data in the input. For example,

```
PRINT, FORMAT = '("First", 15X, "Last")'
```

or

```
PRINT, FORMAT = '("First", TR15, "Last")'
```

results in the following output:

```
FirstbbbbbbbbbbbbbbbbLast
```

where “*b*” represents a blank space.

These two format codes differ in one way. Using the X format code at the end of an output record will not cause any characters to be written unless it is followed by another format code that causes characters to be output. The TR format code always writes characters in this situation. Thus,

```
PRINT, FORMAT = '("First", 15X)'
```

does not leave 15 blanks at the end of the line, but the following statement does:

```
PRINT, FORMAT = ('First', 15TR)'
```

## “C()” Format Code

The C() format code is used to transfer calendar (Julian date and/or time) data. The format is

```
[n]C([c0, c1, ..., cx])
```

where:

*n* — is an optional repeat count ( $1 \leq n \leq 32767$ ) specifying the number of times the format code should be processed. If *n* is not specified, a repeat count of 1 is used.

*c<sub>i</sub>* — represents optional calendar format subcodes, or any of the standard format codes that are allowed within a calendar specification, as described below. If no *c<sub>i</sub>* are provided, the data will be transferred using the standard 24-character system format that includes the day, date, time, and year, as shown in this string:

```
Thu Aug 13 12:01:32 1979
```

This default is equivalent (for output) to:

```
C(CDWA, X, CMOA, X, CDI, X, CHI, ":", CMI, ":", CSI, X, CYI)
```

---

### Note

The C() format code represents an atomic data transfer. Nesting within the parentheses is not allowed.

---

## Calendar Format Subcodes

The following is a list of the subcodes allowed within the parenthesis of the C format code:

### “CMOA” subcodes

The CMOA subcodes transfers the month portion of a date as a string. The format for an all upper case month string is:

```
CMOA[w]
```

The format for a capitalized month string is:

```
CMoA[w]
```

The format for an all lower case month string is:

```
CmoA[w]
```



**Note**

The case of the ‘M’ and ‘O’ of these subcodes will be ignored on input, or if the MONTHS keyword for the current routine is explicitly set.

For these subcodes:

$w$  — is an optional width ( $0 \leq w \leq 256$ ) specifying the number of characters of the month name to be transferred. If  $w$  is not specified, three characters will be transferred. If  $w$  is 0, the natural length of the month name is transferred. On output, if  $w$  is greater than the natural length of the month name, the string will be right justified.

**“CMOI” subcode**

The CMOI subcode transfers the month portion of a date as an integer. The format is as follows:

```
CMOI[w] or CMOI[w.m]
```

where:

$w$  — is an optional width ( $1 \leq w \leq 256$ ) specifying the width of the field in characters. The default width is 2.

$m$  — On output,  $m$  specifies the minimum number of nonblank digits required ( $1 \leq m \leq 256$ ). The field is zero-filled on the left if necessary. If  $m$  is omitted or zero, the external field is blank filled.

**“CDI” subcode**

The CDI subcode transfers the day portion of a date as an integer. The format is as follows:

```
CDI[w] or CDI[w.m]
```

where:

$w$  — is an optional width ( $1 \leq w \leq 256$ ) specifying the width of the field in characters. The default width is 2.

$m$  — On output,  $m$  specifies the minimum number of nonblank digits required ( $1 \leq m \leq 256$ ). The field is zero-filled on the left if necessary. If  $m$  is omitted or zero, the external field is blank filled.

**“CYI” subcode**

The CYI subcode transfers the year portion of a date as an integer. The format is as follows:

```
CYI[w] or CYI[w.m]
```

where:

$w$  — is an optional width ( $1 \leq w \leq 256$ ) specifying the width of the field in characters. The default width is 4.

$m$  — On output,  $m$  specifies the minimum number of nonblank digits required ( $1 \leq m \leq 256$ ). The field is zero-filled on the left if necessary. If  $m$  is omitted or zero, the external field is blank filled.

**“CHI” subcodes**

The CHI subcodes transfer the hour portion of a date as an integer. The format for 24 hour based integer is:

```
CHI[w] or CHI[w.m]
```

The format for a 12 hour based integer is:

```
ChI[w] or ChI[w.m]
```

For these subcodes:

$w$  — is an optional width ( $1 \leq w \leq 256$ ) specifying the width of the field in characters. The default width is 2.

$m$  — On output,  $m$  specifies the minimum number of nonblank digits required ( $1 \leq m \leq 256$ ). The field is zero-filled on the left if necessary. If  $m$  is omitted or zero, the external field is blank filled.

**“CMI” subcode**

The CMI subcode transfers the minute portion of a date as an integer. The format is as follows:

```
CMI[w] or CMI[w.m]
```

where:

$w$  — is an optional width ( $1 \leq w \leq 256$ ) specifying the width of the field in characters. The default width is 2.

$m$  — On output,  $m$  specifies the minimum number of nonblank digits required ( $1 \leq m \leq 256$ ). The field is zero-filled on the left if necessary. If  $m$  is omitted or zero, the external field is blank filled.

### “CSI” subcode

The CSI subcode transfers the seconds portion of a date as an integer. The format is as follows:

```
CSI[w] or CSI[w.m]
```

where:

$w$  — is an optional width ( $1 \leq w \leq 256$ ) specifying the width of the field in characters. The default width is 2.

$m$  — On output,  $m$  specifies the minimum number of nonblank digits required ( $1 \leq m \leq 256$ ). The field is zero-filled on the left if necessary. If  $m$  is omitted or zero, the external field is blank filled.

### “CSF” subcode

The CSF subcode transfers the seconds portion of a date as a floating point value. The format is as follows:

```
CSF[w.d]
```

where:

$w.d$  — is an optional width specification ( $1 \leq w \leq 256$ ,  $1 \leq d < w$ ). The variable  $w$  specifies the number of characters in the external field; the default is 5. The variable  $d$  specifies the number of positions after the decimal point; the default is 2. The value of  $w$  must be large enough to include at least one digit to the left of the decimal point, the decimal point, and  $d$  digits to the right of the decimal point. On output, if the field provided is not wide enough, it is filled with asterisks (\*) to indicate the overflow condition. If  $w$  is zero, the “natural” width for the value is used – the value is read or output using a default format without any leading or trailing whitespace, in the style of the C standard library printf (3S) function.

### “CDWA” subcodes

The CDWA subcodes transfers the day of week portion of a data as a string. The format for an all upper case day of week string is:

```
CDWA[w]
```

The format for a capitalized day of week string is:

```
CDwA[w]
```

The format for an all lower case day of week string is:

```
CdwA[w]
```

---

**Note**

The case of the ‘D’ and ‘W’ of these subcodes will be ignored on input, or if the DAYS\_OF\_WEEK keyword for the current routine is explicitly set.

---

For these subcodes:

$w$  — is an optional width ( $0 \leq w \leq 256$ ), specifying the number of characters of the day of week name to be transferred. If  $w$  is not specified, three characters will be transferred. If  $w$  is 0, the natural length of the day of week name is transferred. On output, if  $w$  is greater than the natural length of the day of week name, the string will be right justified.

**“CAPA” subcodes**

The CAPA subcodes transfers the am or pm portion of a date as a string. The format for an all upper case AM or PM string is:

```
CAPA[w]
```

The format for a capitalized AM or PM string is:

```
CApA[w]
```

The format for an all lower case AM or PM string is:

```
CapA[w]
```

---

**Note**

The case of the first ‘A’ and ‘P’ of these subcodes will be ignored on input, or if the AM\_PM keyword for the current routine is explicitly set.

---

For these subcodes:

$w$  — is an optional width ( $0 \leq w \leq 256$ ), specifying the number of characters of the AM or PM string to be transferred. If  $w$  is not specified, two characters will be transferred. If  $w$  is 0, the natural length of the AM or PM string is transferred. On output, if  $w$  is greater than the natural length of the AM or PM string, the string will be right justified.

## Standard Format Codes Allowed within a Calendar Specification

None of these subcodes are allowed outside of a C() format specifier. In addition to the subcodes listed above, only quoted strings and “X” format codes are allowed inside of the C() format specifier.

### Example:

To print the current date in the default format:

```
PRINT, FORMAT='(C())', SYSTIME(/JULIAN)
```

The printed result should look something like:

```
Fri Aug 14 12:34:14 1998
```

### Example:

To print the current date as a two-digit month value followed by a slash followed by a two-digit day value:

```
PRINT, FORMAT='(C(CMOI, "/ ", CDI))', SYSTIME(/JULIAN)
```

The printed result should look something like:

```
8/14
```

## Example: Reading Tables of Formatted Data

IDL explicitly formatted input/output has the power and flexibility to handle almost any kind of formatted data. A common use of explicitly formatted input/output involves reading and writing tables of data. Consider a data file containing employee data records. Each employee has a name (String, 32 columns) and the number of years they have been employed (Integer, 3 columns) on the first line. The next two lines contain each employee’s monthly salary for the last twelve months. A sample file named *employee.dat* with this format might look like the following:

Bullwinkle					10
1000.0	9000.97	1100.0			2000.0
5000.0	3000.0	1000.12	3500.0	6000.0	900.0
Boris					11
400.0	500.0	1300.10	350.0	745.0	3000.0
200.0	100.0	100.0	50.0	60.0	0.25
Natasha					10
950.0	1050.0	1350.0	410.0	797.0	200.36
2600.0	2000.0	1500.0	2000.0	1000.0	400.0
Rocky					11
1000.0	9000.0	1100.0	0.0	0.0	2000.37
5000.0	3000.0	1000.01	3500.0	6000.0	900.12

The following IDL statements read data with the above format and produce a summary of the contents of the file:

```

;Open data file for input.
OPENR, 1, 'employee.dat'

;Create variables to hold the name, number of years, and monthly
;salary.
name = '' & years = 0 & salary = FLTARR(12)

;Output a heading for the summary.
PRINT, FORMAT=('Name", 28X, "Years", 4X, "Yearly Salary")'

;Note: The actual dashed line is longer than is shown here.
PRINT, '======'

;Loop over each employee.
WHILE (NOT EOF(1)) DO BEGIN

    ;Read the data on the next employee.
    READF, 1, $
    FORMAT = '(A32,I3,2(/,6F10.2))', name, years, salary

;Output the employee information. Use TOTAL to sum the monthly
;salaries to get the yearly salary.
    PRINT, FORMAT='(A32,I5,5X,F10.2)', name, years, TOTAL(salary)

ENDWHILE

CLOSE, 1

```

The output from executing these statements on *employee.dat* is as follows:

Name	Years	Yearly Salary
=====		
Bullwinkle	10	32501.09
Borris	11	6805.35
Natasha	10	14257.36
Rocky	11	32500.50

## Example: Reading Records that Contain Multiple Array Elements

Frequently, data are written to files with each record containing single elements of more than one array. One example might be a file consisting of observations of altitude, pressure, temperature, and velocity with each line or record containing a value for each of the four variables. Because IDL has no equivalent of the FORTRAN implied DO list, special procedures must be used to read or write this type of file.

The first approach, which is the simplest, may be used only if all of the variables have the same data type. An array is created with as many columns as there are variables and as many rows as there are elements. The data are read into this array, the array is transposed storing each variable as a row, and each row is extracted and stored into a variable which becomes a vector. For example, the FORTRAN program which writes the data and the IDL program which reads the data are as follows:

### **FORTRAN Write:**

```
DIMENSION ALT(100),PRES(100),TEMP(100),VELO(100)
OPEN (UNIT = 1, STATUS='NEW', FILE='TEST')
WRITE(1, '(4(1x,g15.5))')
      (ALT(I),PRES(I),TEMP(I),VELO(I),I=1,100)
```

### **IDL Read:**

```
;Open file for input.
OPENR, 1, 'test'

;Define variable (NVARs by NOBS).
A = FLTARR(4,100)

;Read the data.
READF, 1, A

;Transpose so that columns become rows.
A = TRANSPOSE(A)

;Extract the variables.
ALT = A[* , 0]
PRES = A[* , 1]
TEMP = A[* , 2]
VELO = A[* , 3]
```

Note that this same example may be written without the implied DO list, writing all elements for each variable contiguously and simplifying matters considerably:

### **FORTRAN Write:**

```
DIMENSION ALT(100),PRES(100),TEMP(100),VELO(100)
OPEN (UNIT = 1, STATUS='NEW', FILE='TEST')
WRITE (1, '(4(1x,G15.5))') ALT,PRES,TEMP,VELO
```

### **IDL Read:**

```
;Define variables.
ALT = FLTARR(100)
PRES = ALT & TEMP = ALT & VELO = ALT
OPENR, 1, 'test'
READF, 1, ALT, PRES, TEMP, VELO
```

A different approach must be taken when the columns contain different data types or the number of lines or records are not known. This method involves defining the arrays, defining a scalar variable to contain each datum in one record, then writing a loop to read each line into the scalars, and then storing the scalar values into each array. For example, assume that a fifth variable, the name of an observer which is of string type, is added to the variable list. The FORTRAN output routine and IDL input routine are as follows:

### **FORTRAN Write:**

```
DIMENSION ALT(100),PRES(100),TEMP(100),VELO(100)
CHARACTER * 10 OBS(100)
OPEN (UNIT = 1, STATUS = 'NEW', FILE = 'TEST')
WRITE (1, '(4(1X,G15.5),2X,A)')
      (ALT(I),PRES(I),TEMP(I),VELO(I),OBS(I),I=1,100)
```

### **IDL Read:**

```
;Access file. This example will read files containing from 1 to 200
;records.
OPENR, 1, 'test'

;Define vector, make it large enough for the biggest case.
ALT = FLTARR(200)

;Define other vectors using the first.
PRES = ALT & TEMP = ALT & VELO = ALT

;Define string array.
OBS = STRARR(200)

;Define scalar string.
OBSS = ''

;Initialize counter.
I = 0

;Read loop.
WHILE NOT EOF(1) DO BEGIN

    ;Read scalars.
    READF, 1, $

    FORMAT = '(4(1X, G15.5), 2X, A10)', $
        ALTS, PRESS, TEMPS, VELOS, OBSS

;Store in each vector.
    ALT[I] = ALTS & PRES[I] = PRESS & TEMP[I] = TEMPS
```



```

      VELO[I] = VELOS & OBS[I] = OBSS

      ;Increment counter and check for too many records.
      IF I LT 199 THEN I = I + 1 ELSE STOP, 'Too many records'

      ;Done.
      ENDWHILE

```

If desired, after the file has been read and the number of observations is known, the arrays may be truncated to the correct length using a series of statements similar to the following:

```

      ALT = ALT[0:I-1]

```

The above statement represents a worst case example. Reading is greatly simplified by writing data of the same type contiguously and by knowing the size of the file. One frequently used technique is to write the number of observations into the first record so that when reading the data the size is known.

### Warning

---

It might be tempting to implement a loop in IDL which reads the data values directly into array elements, using a statement such as the following:

```

FOR I = 0, 99 DO READF, 1, ALT[I], PRES[I], TEMP[I], VELO[I]

```

This statement is *incorrect*. Subscripted elements (including ranges) are temporary expressions passed as values to procedures and functions (READF in this example). Parameters passed by value do not pass results back to the caller. The proper approach is to read the data into scalars and assign the values to the individual array elements as follows:

```

A = 0. & P = 0. & T = 0. & V = 0.
FOR I = 0, 99 DO BEGIN
    READF, 1, A, P, T, V
    ALT[I] = A & PRES[I] = P & TEMP[I] = T & VELO[I] = V
ENDFOR

```

---

# Using Unformatted Input/Output

Unformatted input/output involves the direct transfer of data between a file and memory without conversion to and from a character representation. Unformatted input/output is used when efficiency is important and portability is not an issue. It is faster and requires less space than formatted input/output. IDL provides three procedures for performing unformatted input/output:

## READU

Reads unformatted data from the specified file unit.

## WRITEU

Writes unformatted data to the specified file unit.

## ASSOC

Maps an array structure to a logical file unit, providing efficient and convenient direct access to data.

This section discusses READU and WRITEU, while ASSOC is discussed in [“Associated Input/Output”](#) on page 406. The READU and WRITEU procedures provide IDL’s basic unformatted input/output capabilities. They have the form:

```
READU, Unit, Var1, ..., Varn
WRITEU, Unit, Var1, ..., Varn
```

where

*Unit* — The logical file unit with which the input/output operation will be performed.

*Var<sub>i</sub>* — One or more IDL variables (or expressions in the case of output).

The WRITEU procedure writes the contents of its arguments directly to the file, and READU reads exactly the number of bytes required by the size of its arguments. Both cases directly transfer binary data with no interpretation or formatting.

## Unformatted Input/Output of String Variables

Strings are the only basic IDL data type that do not have a fixed size. A string variable has a dynamic length that is dependent only on the length of the string currently assigned to it. Thus, although it is always possible to know the length of the other types, string variables are a special case. IDL uses the following rules to determine the number of characters to transfer:

## Input

Input enough bytes to fill the original length of the string. The length of the resulting string is truncated if the string contains a null byte.

## Output

Output the number of bytes contained in the string. This number is the same number returned by the STRLEN function and does not include a terminating null byte.

Note that these rules imply that when reading into a string variable from a file, you must know the length of the original string so as to be able to initialize the destination string to the correct length. For example, the following IDL statements produce the following output, because the receiving variable A was not long enough.

```

;Open a file.
OPENW, 1, 'temp.tmp'

;Write an 11-character string.
WRITEU, 1, 'Hello World'

;Rewind the file.
POINT_LUN, 1, 0

;Prepare a nine-character string.
A = '          '

;Read back in the string.
READU, 1, A

;Show what was input.
PRINT, A

CLOSE, 1

```

produce the following, because the receiving variable A was not long enough:

```
Hello Wor
```

The only solution to this problem is to know the length of the string being input. The following IDL statements demonstrate a useful “trick” for initializing strings to a known length:

```

;Open a file.
OPENW, 1, 'temp.tmp'

;Write an 11-character string.
WRITEU, 1, 'Hello World'

```

```

;Rewind the file.
POINT_LUN, 1, 0

;Create a string of the desired length initialized with blanks.
;REPLICATE creates a byte array of 11 elements, each element
;initialized to 32, which is the ASCII code for a blank. Passing
;this byte array to STRING converts it to a scalar string
;containing 11 blanks.
A = STRING(REPLICATE(32B,11))

;Read in the string.
READU, 1, A

;Show what was input.
PRINT, A

CLOSE, 1

```

This example takes advantage of the special way in which the `BYTE` and `STRING` functions convert between byte arrays and strings. See the description of the `BYTE` and `STRING` functions for additional details.

## Example: Reading C-Generated Unformatted Data with IDL

The following C program produces a file containing employee records. Each record stores the first name of each employee, the number of years he has been employed, and his salary history for the last 12 months.

```

#include <stdio.h>

main()
{
    static struct rec {
        char name[32]; /* Employee's name */
        int years;    /* # of years with company */
        float salary[12]; /* Salary for last 12 months */
    } employees[] = {
    { {'B','u','l','l','w','i','n','k','l','e'}, 10,
      {1000.0, 9000.97, 1100.0, 0.0, 0.0, 2000.0,
       5000.0, 3000.0, 1000.12, 3500.0, 6000.0, 900.0} },{
    {'B','o','r','r','i','s'}, 11,
      {400.0, 500.0, 1300.10, 350.0, 745.0, 3000.0,
       200.0, 100.0, 100.0, 50.0, 60.0, 0.25} },
    { {'N','a','t','a','s','h','a'}, 10,
      {950.0, 1050.0, 1350.0, 410.0, 797.0, 200.36,
       2600.0, 2000.0, 1500.0, 2000.0, 1000.0, 400.0} },
    { {'R','o','c','k','y'}, 11,
      {1000.0, 9000.0, 1100.0, 0.0, 0.0, 2000.37,

```

```

        5000.0, 3000.0, 1000.01, 3500.0, 6000.0, 900.12}}
};

FILE *outfile;

outfile = fopen("data.dat", "w");
(void) fwrite(employees, sizeof(employees), 1, outfile);
(void) fclose(outfile);
}

```

Running this program creates the file *data.dat* containing the employee records. The following IDL statements can be used to read and print this file:

```

;Create a string with 32 characters so that the proper number of
;characters will be input from the file. REPLICATE is used to
;create a byte array of 32 elements, each containing the ASCII code
;for a space (32). STRING turns this byte array into a string
;containing 32 blanks.
STR32 = STRING(REPLICATE(32B, 32))

;Create an array of four employee records to receive the input
;data.
A = REPLICATE({EMPLOYEES, NAME:STR32, YEARS:0L, $
              SALARY:FLTARR(12)}, 4)

;Open the file for input.
OPENR, 1, 'data.dat'

;Read the data.
READU, 1, A

CLOSE, 1

;Show the results.
PRINT, A

```

Executing these IDL statements produces the following output:

```

{ Bullwinkle          10
1000.00    9000.97    1100.00    0.00000    0.00000    2000.00
5000.00    3000.00    1000.12    3500.00    6000.00    900.000
}{Borris              11
400.000    500.000    1300.10    350.000    745.000    3000.00
200.000    100.000    100.000    50.00000    60.00000    0.250000
}{ Natasha            10
950.000    1050.00    1350.00    410.000    797.000    200.360
2600.00    2000.00    1500.00    2000.00    1000.00    400.000
}{ Rocky              11
1000.00    9000.00    1100.00    0.00000    0.00000    2000.37
5000.00    3000.00    1000.01    3500.00    6000.00    900.120

```

```
}
```

## Example: Reading IDL-Generated Unformatted Data with C

The following IDL program creates an unformatted data file containing a 5 x 5 array of floating-point values:

```
;Open a file for output.
OPENW, 1, 'data.dat'

;Write 5x5 array with each element set to its 1-dimensional index.
WRITEU, 1, FINDGEN(5, 5)

CLOSE, 1
```

This file can be read and printed by the following C program:

```
#include <stdio.h>

main()
{
    float data[5][5];
    FILE *infile; int i, j;
    infile = fopen("data.dat", "r");
    (void) fread(data, sizeof(data), 1, infile);
    (void) fclose(infile);
    for (i = 0; i < 5; i++) {
        for (j = 0; j < 5; j++)
            printf("%8.1f", data[i][j]);
        printf("\n");
    }
}
```

Running this program gives the following output:

```
0.0    1.0    2.0    3.0    4.0
5.0    6.0    7.0    8.0    9.0
10.0   11.0   12.0   13.0   14.0
15.0   16.0   17.0   18.0   19.0
20.0   21.0   22.0   23.0   24.0
```

## Example: Reading a Sun Rasterfile from IDL

Sun computers use rasterfiles to store scanned images. This example shows how to read such an image and display it using IDL. In the interest of keeping the example brief, a number of simplifications are made, no error checking is performed, and only 8-bit deep rasterfiles are handled. See the READ\_SRF procedure (the file read\_srf.pro in the lib subdirectory of the IDL distribution) for a complete example.

The format used for rasterfiles is documented in the C header file `/usr/include/rasterfile.h`. That file provides the following information:

Each file starts with a fixed header that describes the image. In C, this header is defined as follows:

```
struct rasterfile{
    int ras_magic; /* magic number */
    int ras_width; /* width (pixels) of image */
    int ras_height; /* height (pixels) of image */
    int ras_depth; /* depth (1, 8, or 24 bits) */
    int ras_length; /* length (bytes) of image */
    int ras_type; /* type of file */
    int ras_maptype; /* type of colormap */
    int ras_maplength; /* length(bytes) of colormap */ };
```

The color map, if any, follows directly after the header information. The image data follows directly after the color map.

The following IDL statements read an 8-bit deep image from the file *ras.dat*:

```
;Define IDL structure that matches the Sun-defined rasterfile
;structure. A C int variable on a Sun corresponds to an IDL LONG
;int.
h = {rasterfile, magic:0L, width:0L, height:0L, depth: 0L,$
     length:0L, type:0L, maptype:0L, maplength:0L}

;Open the file, allocating a file unit at the same time.
OPENR, unit, file, /GET_LUN

;Read the header information.
READU, unit, h

;Is there a color map?
IF ((h.maptype EQ 1) AND (h.maplength NE 0) ) THEN BEGIN

    ;Calculate length of each vector.
    maplen = h.maplength/3

    ;Create three byte vectors to hold the color map.
    r=(g=(b=BYTARR(maplen, /NOZERO)))

    ;Read the color map.
    READU, unit, r, g, b

ENDIF

;Create a byte array to hold image.
image = BYTARR(h.width, h.height, /NOZERO)
```

```
;Read the image.  
READU, unit, image  
  
;Free the previously-allocated Logical Unit Number and close the  
;file.  
FREE_LUN, unit
```



# Portable Unformatted Input/Output

Normally, unformatted input/output is not portable between different machine architectures because of differences in the way various machines represent binary data. However, it is possible to produce binary files that are portable by specifying the XDR keyword with the OPEN procedures. XDR (for eXternal Data Representation) is a scheme under which all binary data is written using a standard “canonical” representation. All machines supporting XDR understand this standard representation and have the ability to convert between it and their own internal representation.

XDR represents a compromise between the extremes of unformatted and formatted input/output:

- It is not as efficient as purely unformatted input/output because it does involve the overhead of converting between the external and internal binary representations.
- It is still much more efficient than formatted input/output because conversion to and from ASCII characters is much more involved than converting between binary representations.
- It is much more portable than purely unformatted data, although it is still limited to those machines that support XDR. However, XDR is freely available and can be moved to any system.

## XDR Considerations

The primary differences in the way IDL input/output procedures work with XDR files, as opposed to files opened normally are as follows:

- To use XDR, you must specify the XDR keyword when opening the file.
- The only input/output data transfer routines that can be used with a file opened for XDR are READU and WRITEU.
- XDR converts between the internal and standard external binary representations for data instead of simply using the machine’s internal representation.
- Since XDR adds extra “bookkeeping” information to data stored in the file and because the binary representation used may not agree with that of the machine being used, it does not make sense to access an XDR file without using XDR.

- OPENW and OPENU normally open files for both input and output. However, XDR files can only be opened in one direction at a time. Thus, using these procedures with the XDR keyword results in a file open for output only. OPENR works in the usual way.
- The length of strings is saved and restored along with the string. This means that you do not have to initialize a string of the correct length before reading a string from the XDR file. (This is necessary with normal unformatted input/output and is described in “Using Unformatted Input/Output” on page 394).
- For efficiency reasons, byte arrays are transferred as a single unit; therefore, byte variables must be initialized to the correct number of elements for the data to be input, or an error will occur. For example, given the statements,

```

;Open a file for XDR output.
OPENW, /XDR, 1, 'data.dat'

;Write a 10-element byte array.
WRITEU, 1, BINDGEN(10)

;Close the file and re-open it for input.
CLOSE, 1 & OPENR, /XDR, 1, 'data.dat'

```

then the statement,

```

;Try to read the first byte only.
B = 0B & READU, 1, B

```

results in the following error:

```

% READU: Error encountered reading from file unit: 1.

```

Instead, it is necessary to read the entire byte array back in one operation using a statement such as:

```

;Read the whole array back at once.
B=BYTARR(10) & READU, 1, B

```

This restriction does not exist for other data types.

- Under VMS, XDR is only possible with stream mode files.

## IDL XDR Conventions for Programmers

IDL uses certain conventions for reading and writing XDR files. If your only use of XDR is through IDL, you do not need to be concerned about these conventions because IDL takes care of it for you. However, programmers who want to create IDL-

compatible XDR files from other languages need to know the actual XDR routines used by IDL for various data types. The following table summarizes this information.

Data Type	XDR routine
Byte	xdr_bytes()
Integer	xdr_short()
Long	xdr_long()
Float	xdr_float()
Double	xdr_double()
Complex	xdr_complex()
String	xdr_counted_string()

*Table 16-10: XDR Routines Used by IDL*

The routines used for type COMPLEX and STRING are not primitive XDR routines. Their definitions are as follows:

```

bool_t xdr_complex(xdrs, p)
    XDR *xdrs;
    struct complex { float r, i } *p;
{
    return(xdr_float(xdrs, (char *) &p->r) &&
           xdr_float(xdrs, (char *) &p->i));
}
bool_t xdr_counted_string(xdrs, p)
    XDR *xdrs;
    char **p;
{
    int input = (xdrs->x_op == XDR_DECODE);
    short length;

    /* If writing, obtain the length */
    if (!input) length = strlen(*p);

    /* Transfer the string length */
    if (!xdr_short(xdrs, (char *) &length)) return(FALSE);

    /* If reading, obtain room for the string */
    if (input)
    {
        *p = malloc((unsigned) (length + 1));
    }
}

```

```

        *p[length] = '\\0'; /* Null termination */
    }
    /* If the string length is nonzero, transfer it */
    return(length ? xdr_string(xdrs, p, length) : TRUE);
}

```

## Example: Reading C-Generated XDR Data with IDL

The following C program produces a file containing different types of data using XDR. The usual error checking is omitted for the sake of brevity.

```

#include <stdio.h>
#include <rpc/rpc.h>
[ xdr_complex() and xdr_counted_string() included here ]

main()
{
    static struct { /* Output data */
        unsigned char c;
        short s;
        long l;
        float f;
        double d;
        struct complex { float r, i } cmp;
        char *str;
    }
    data = {1, 2, 3, 4, 5.0, { 6.0, 7.0}, "Hello" };
    u_int c_len = sizeof(unsigned char); /* Length of a char */
    char *c_data = (char *) &data.c;    /* Addr of byte field */
    FILE *outfile;                       /* stdio stream ptr */
    XDR xdrs;                             /* XDR handle */

    /* Open stdio stream and XDR handle */
    outfile = fopen("data.dat", "w");
    xdrstdio_create(&xdrs, outfile, XDR_ENCODE);

    /* Output the data */
    (void) xdr_bytes(&xdrs, &c_data, &c_len, c_len);
    (void) xdr_short(&xdrs, (char *) &data.s);
    (void) xdr_long(&xdrs, (char *) &data.l);
    (void) xdr_float(&xdrs, (char *) &data.f);
    (void) xdr_double(&xdrs, (char *) &data.d);
    (void) xdr_complex(&xdrs, (char *) &data.cmp);
    (void) xdr_counted_string(&xdrs, &data.str);

    /* Close XDR handle and stdio stream */
    xdr_destroy(&xdrs);
    (void) fclose(outfile);
}

```

Running this program creates the file *data.dat* containing the XDR data. The following IDL statements can be used to read this file and print its contents:

```
;Create structure containing correct types.
DATA={S, C:0B, S:0, L:0L, F:0.0, D:0.0D, CMP:COMPLEX(0), STR:''}

;Open the file for input.
OPENR, /XDR, 1, 'data.dat'

;Read the data.
READU, 1, DATA

;Close the file.
CLOSE, 1

;Show the results.
PRINT, DATA
```

Executing these IDL statements produces the output:

```
{ 1      2      3      4.00000      5.0000000
 (      6.00000,      7.00000) Hello}
```

For further details about XDR, consult the XDR documentation for your machine. Sun users should consult their *Network Programming* manual.

# Associated Input/Output

Unformatted data stored in files often consists of a repetitive series of arrays or structures. A common example is a series of images. IDL-associated file variables offer a convenient and efficient way to access such data.

An associated variable is a variable that maps the structure of an IDL array or structure variable onto the contents of a file. The file is treated as an array of these repeating units of data. The first array or structure in the file has an index of zero, the second has index one, and so on. Such variables do not keep data in memory like a normal variable. Instead, when an associated variable is subscripted with the index of the desired array or structure within the file, IDL performs the input/output operation required to access the data.

When their use is appropriate (the file consists of a sequence of identical arrays or structures), associated file variables offer the following advantages over `READU` and `WRITEU` for unformatted input/output:

- Input/output occurs when an associated file variable is subscripted. Thus, it is possible to perform input/output within an expression without a separate input/output statement.
- The size of the data set is limited primarily by the maximum size of the file containing the data instead of the maximum memory available. Data sets too large for memory can be accessed.
- There is no need to declare the maximum number of arrays or structures contained in the file.
- Associated variables systematize access to the data. Direct access to any element in the file is rapid and simple—there is no need to calculate offsets into the file and/or position the file pointer prior to performing the input/output operation.
- Associated variables are the most efficient form of IDL input/output.

An associated file variable is created by assigning the result of the `ASSOC` function to a variable. See [ASSOC](#) in the *IDL Reference Guide* for details.

## Example of Using Associated Input/Output

Assume that a file named *data.dat* exists, and that this file contains a series of 10 x 20 arrays of floating-point data. The following two IDL statements open the file and create an associated file variable mapped to the file:

```

;Open the file.
OPENU, 1, 'data.dat'

;Make a file variable. Using the NOZERO keyword with FLTARR
;increases efficiency.
A = ASSOC(1, FLTARR(10, 20, /NOZERO))

```

The order of these two statements is not important—it would be equally valid to call ASSOC first, and then open the file. This is because the association is between the variable and the logical file unit, not the file itself. It is also legitimate to close the file, open a new file using the same LUN, and then use the associated variable without first executing a new ASSOC. Naturally, an error occurs if the file is not open when the file variable is subscripted in an expression or if the file is open for the wrong type of access (for example, trying to assign to an associated file variable linked with a file opened for read-only access).

As a result of executing the two statements above, the variable A is now an associated file variable. Executing the statement,

```
HELP, A
```

gives the following response:

```
A          FLOAT      = File<data.dat> Array(10, 20)
```

The associated variable A maps the structure of a 10 x 20, floating-point array onto the contents of the file *data.dat*. Thus, the response from the HELP procedure shows it as having the structure of a two-dimensional array. An associated file variable only performs input/output to the file when it is subscripted. Thus, the following two IDL statements do not cause input/output to happen:

```
B = A
```

This assignment does not transfer data from the file to variable B because A is not subscripted. Instead, B becomes an associated file variable with the same structure, and to the same logical file unit, as A.

```
B = 23
```

This assignment does not result in the value 23 being transferred to the file because variable B (which became a file variable in the previous statement) is not subscripted. Instead, B becomes a scalar integer variable containing the value 23. It is no longer an associated file variable.

## Reading Data from Associated Files

Once a variable has been associated with a file, data are read from the file whenever the associated variable appears in an expression with a subscript. The position of the

array or structure read from the file is given by the value of the subscript. The following IDL statements give some examples of using file variables:

```

;Copy the contents of the first array into normal variable Z. Z is
;now a 10 x 20, floating-point array.
Z = A[0]

;Form the sum of the first 10 arrays (Z was initialized in the
;previous statement to the value of the first array. This statement
;adds the following nine to it.).
FOR I = 1, 9 DO Z = Z + A[I]

;Read fourth array and plot it.
PLOT, A[3]

;Subtract array four from array five, and plot the result. The
;result of the subtraction is then discarded.
PLOT, A[5] - A[4]

```

## Subscripting Associated File Variables on Input

When the structure associated with a file variable is an array, it is possible to subscript into the array being accessed during input operations. For example, for the variable *A* defined above,

```
Z = A[0, 0, 1]
```

assigns the value of the first floating-point element of the second array within the file to the variable *Z*. The rightmost subscript is taken as the subscript into the file causing IDL to read the entire array into memory. This resulting array expression is then further subscripted by the remaining subscripts.

### Note

---

Although this ability can be convenient, it also can be very slow because every access to an array element causes the entire array to be read from memory. Unless only one element of the array is desired, it is faster to assign the contents of the array to a normal variable by subscripting the file variable with a single subscript, then accessing the individual array elements in the normal variable.

---

## Writing Data

When a subscripted associated variable appears on the left side of an assignment statement, the expression on the right side is written into the file at the given array position:



```

;Sets sixth record to zero.
A[5] = FLTARR(10, 20)

;Writes ARR into the sixth record after any necessary type
;conversions.
A[5] = ARR

;Averages records J and J+1, and writes the result into record J.
A[J] = (A[J] + A[J + 1])/2

```

When writing data, only a single subscript specifying the index of the affected array or structure in the file is allowed. Thus, it is not possible to index individual elements of associated arrays on output, although it is allowed for input. To update individual elements of an array within a file, assign the contents of that array to a normal array variable, modify the copy, and write the array back by assigning it to the subscripted file variable.

## Files with Multiple Structures

The same file may be associated with a number of different structures. Assume a number of 128 x 128-byte images are contained on a file. The statement,

```
ROW = ASSOC(1, BYTARR(128))
```

will map the file into rows of 128 bytes each. ROW[3] is the fourth row of the first image, while ROW[128] is the first row of the second image. The statement,

```
IMAGE = ASSOC(1, BYTARR(128, 128))
```

maps the file into entire images; IMAGE[4] will be the fifth image.

## Offset Parameter

The *Offset* parameter to ASSOC specifies the position in the file at which the first array starts. This parameter is useful when a file contains a header followed by data records. For example, if a UNIX file uses the first 1,024 bytes of the file to contain header information, followed by 512 x 512-byte images, the statement,

```
IMAGE = ASSOC(1, BYTARR(512, 512), 1024)
```

sets the variable IMAGE to access the images while skipping the header.

Under VMS, stream files and RMS block mode files have their offset given in bytes, and record-oriented files have it specified in records. Thus, the example above would have worked for VMS if the file was a stream or block mode file. Assume however, that the file has 512-byte, fixed-length records. In this case, skipping the first 1,024 bytes is equivalent to skipping the first two records:

```
IMAGE = ASSOC(1, BYTARR(512, 512), 2)
```

## Efficiency

Arrays are accessed most efficiently if their length is an integer multiple of the physical block size of the disk holding the file. Common values are 512, 1,024, and 2,048 bytes. For example, on a disk with 512-byte blocks, one benchmark program required approximately one-eighth of the time required to read a 512 x 512-byte image that started and ended on a block boundary, as compared to a similar program that read an image that was not stored on even block boundaries.

Each time a subscripted associated variable is referenced, one or more records are read from or written to the file. Therefore, if a record is to be accessed more than a few times, it is more efficient to read the entire record into a variable. After making the required changes to the in-memory variable, it can be written back to the file if necessary.

## Unformatted Data from UNIX FORTRAN Programs

Unformatted data files generated by FORTRAN programs under UNIX contain an extra long word before and after each logical record in the file. ASSOC does not interpret these extra bytes but considers them to be part of the data. This is true even if the F77\_UNFORMATTED keyword is specified on the OPEN statement. Therefore, ASSOC should not be used with such files. Instead, such files should be processed using READU and WRITEU. An example of using IDL to read such data is given in [“Using Unformatted Input/Output”](#) on page 394.

# File Manipulation Operations

## Locating Files

The `FINDFILE` function returns an array of strings containing the names of all files that match its argument string. The argument string may contain any wildcard characters understood by the command interpreter. Under VMS, this is DCL. Under UNIX, it is the Bourne shell (`/bin/sh`). Under Windows it is `COMMAND.COM`. On the Macintosh, standard Macintosh OS wildcard characters are supported. For example, to determine the number of IDL procedure files that exist in the current directory, use the following statement:

```
PRINT, '# IDL pro files:', N_ELEMENTS(FINDFILE('*.*pro'))
```

See [FINDFILE](#) in the *IDL Reference Guide* for details.

## Getting Help and Information

Information about currently open file units is available by using the `FILES` keyword with the `HELP` procedure. If no arguments are provided, information about all currently open user file units (units 1–128) is given. For example, the following command can be used to get information about the three special units (–2, –1, and 0):

```
HELP, /FILES, -2, -1, 0
```

This command results in output similar to the following:

Unit	Attributes	Name
-2	Write, New, Tty, Reserved	<stderr>
-1	Write, New, Tty, Reserved	<stdout>
0	Read, Tty, Reserved	<stdin>

See [HELP](#) in the *IDL Reference Guide* for details.

## The FSTAT Function

The `FSTAT` function can be used to get more detailed information, as well as information that can be used from within an IDL program. It returns a structure expression of type `FSTAT` or `FSTAT64` containing information about the file. For example, to get detailed information about the standard input, the command,

```
HELP, /STRUCTURES, FSTAT(0)
```

causes the following to be displayed on the screen:

```
** Structure FSTAT, 12 tags, length=48:
```

UNIT	LONG	0
NAME	STRING	'<stdin>'
OPEN	BYTE	1
ISATTY	BYTE	1
ISAGUI	BYTE	0
INTERACTIVE	BYTE	1
READ	BYTE	1
WRITE	BYTE	0
TRANSFER_COUNT	LONG	0
CUR_PTR	LONG	51550
SIZE	LONG	0
REC_LEN	LONG	0

Since IDL allows keywords to be abbreviated to the shortest nonambiguous number of characters,

```
HELP, /ST, FSTAT(0)
```

also will work (and save some typing).

IDL on some platforms can support files that are longer than  $2^{31}-1$  bytes in length. If FSTAT is applied to such a file, it returns an expression of type FSTAT64 instead of the FSTAT structure shown above. FSTAT64 differs from FSTAT only in that the TRANSFER\_COUNT, CUR\_PTR, SIZE, and REC\_LEN fields are signed 64-bit integers (type LONG64) in order to be able to represent the larger sizes.

The fields of the FSTAT and FSTAT64 structures provide the following information:

## UNIT

The IDL logical unit number (LUN).

## NAME

The name of the file.

## OPEN

Nonzero if the file unit is open. If OPEN is zero, the remaining fields in FSTAT will not contain useful information.

## ISATTY

Nonzero if the file is actually a terminal instead of a normal file. For example, if you are using an `xterm` window on a Unix system and you invoke FSTAT on logical unit 0 (standard input), ISATTY will be set to 1.

## ISAGUI

Nonzero if the file is actually a Graphical User Interface (for example, a logical unit associated with the IDL Development Environment). Thus, if you are using IDLDE and you invoke FSTAT on logical unit 0 (standard input), ISAGUI will be set to 1.

## INTERACTIVE

Nonzero if *either* ISATTY or ISAGUI is nonzero.

## READ

Nonzero if the file is open for read access.

## WRITE

Nonzero if the file is open for write access.

## TRANSFER\_COUNT

The number of scalar IDL data items transferred in the last input/output operation on the unit. This is set by the following IDL routines: READU, WRITEU, PRINT, PRINTF, READ, and READF. TRANSFER\_COUNT is useful when attempting to recover from input/output errors.

## CUR\_PTR

The current position of the file pointer, given in bytes from the start of the file. If the device is a terminal (ISATTY is nonzero), the value of CUR\_PTR will not contain useful information.

## SIZE

The current length of the file in bytes. If the device is a terminal (ISATTY is nonzero), the value of SIZE will not contain useful information.

## REC\_LEN

If the file is record-oriented (VMS), this field contains the record length; otherwise, it is zero.

## An Example Using FSTAT

The following IDL function can be used to read single-precision, floating-point data from a stream file into a vector when the number of elements in the file is not known. It uses the FSTAT function to get the size of the file in bytes and divides by four (the size of a single-precision, floating-point value) to determine the number of values. Note that this approach will not work with VMS variable-length record files:

```

;READ_DATA reads all the floating point values from a stream file
;and returns the result as a floating-point vector.
FUNCTION READ_DATA, file

;Get a unique file unit and open the data file.
OPENR, /GET_LUN, unit, file

;Get file status.
status = FSTAT(unit)

;Make an array to hold the input data. The SIZE field of status
;gives the number of bytes in the file, and single-precision,
;floating-point values are four bytes each.
data = FLTARR(status.size / 4)

;Read the data.
READU, unit, data

;Deallocate the file unit. The file also will be closed.
FREE_LUN, unit

RETURN, data

END

```

Assuming that a file named *data.dat* exists and contains 10 floating-point values, the `READ_DATA` function could be used as follows:

```

;Read floating-point values from data.dat.
A = READ_DATA('data.dat')

;Show the result.
HELP, A

```

The following output is produced:

```

A                FLOAT      = Array(10)

```

## Flushing File Units

For efficiency, IDL buffers its input/output in memory. Therefore, when data are output, there is a window of time during which data are in memory and have not been actually placed into the file. Normally, this behavior is transparent to the user (except for the improved performance). The `FLUSH` routine exists for those rare occasions where a program needs to be certain that the data has actually been written to the file immediately. For example, use the statement,

```

FLUSH, 1

```

to flush file unit one.

See [FLUSH](#) in the *IDL Reference Guide* for details.

## Positioning File Pointers

Each open file unit has a current file pointer associated with it. This file pointer indicates the position in the file at which the next input/output operation will take place. The file position is specified as the number of bytes from the start of the file. The first position in the file is position zero. The following statement will rewind file unit 1 to its start:

```
POINT_LUN, 1, 0
```

The following sequence of statements will position it at the end of the file:

```
tmp = FSTAT(1)
POINT_LUN, 1, tmp.size
```

`POINT_LUN` has the following operating-system specific behavior:

- **UNIX:** the current file pointer can be positioned arbitrarily – moving to a position beyond the current end-of-file causes the file to grow out to that point. The gap created is filled with zeroes.
- **VMS stream files:** the current file pointer can be positioned arbitrarily – moving to a position beyond the current end-of-file causes the file to grow out to that point. The gap created is filled with zeroes.
- **VMS block mode and record-oriented files:** attempting to move the pointer past the current end-of-file causes an end-of-file error.
- **VMS record-oriented files:** the file pointer should only be set to record boundaries. Setting it to other positions can result in unexpected behavior.
- **Windows:** the current file pointer can be positioned arbitrarily – moving to a position beyond the current end-of-file causes the file to grow out to that point. Unlike UNIX, the gap created is filled with arbitrary data instead of zeroes.
- **Macintosh:** the current file pointer cannot be positioned past the end of the file.

See [POINT\\_LUN](#) in the *IDL Reference Guide* for details.

## Testing for End-Of-File

The EOF function is used to test a file unit to see if it is currently positioned at the end of the file. It returns true (1) if the end-of-file condition is true and false (0) otherwise.

Note the non-diskfile devices always return “false” and, under VMS, non-sequential files or files opened across DECnet always return “false”.

For example, to read the contents of a file and print it on the screen, use the following statements:

```
;Open file demo.doc for reading.
OPENR, 1, 'demo.doc'

;Create a variable of type string.
LINE = ''

;Read and print each line until the end of the file is encountered.
WHILE(NOT EOF(1)) DO BEGIN READF,1,LINE & PRINT,LINE & END

;Done with the file.
CLOSE, 1
```

See [EOF](#) in the *IDL Reference Guide* for details.

## GET\_KBRD

The GET\_KBRD function returns the next character available from the standard input (IDL file unit zero) as a single character string. It takes a single parameter named WAIT. If WAIT is zero, the function returns the null string if there are no characters in the terminal typeahead buffer. If it is nonzero, the function waits for a character to be typed before returning.

Under Windows, the GET\_KBRD function can be used to return Windows special characters (in addition to the standard keyboard characters). To get a special character, hold down the Alt key and type the character’s ANSI equivalent on the numeric keypad while GET\_KBRD is waiting. Control + *key* combinations are not supported.

See [GET\\_KBRD](#) in the *IDL Reference Guide* for details.

### Example—Using GET\_KBRD

A procedure that updates the screen and exits when the carriage return is typed might appear as follows:



```

;Procedure definition.
PRO UPDATE, ...

;Loop forever.
WHILE 1 DO BEGIN

;Update screen here...
...

;Read character, no wait.
CASE GET_KBRD(0) OF

;Process letter A.
'A': ....

;Process letter B.
'B': ....

;Process other alternatives.
...

;Exit on carriage return (ASCII code = 15 octal).
STRING("15B): RETURN

;Ignore all other characters.
ELSE:

ENDCASE

ENDWHILE

;End of procedure.
END

```

## Using the STRING Function to Format Data

The STRING function is very similar to the PRINT and PRINTF procedures. It can be thought of as a version of PRINT that places its formatted output into a string variable instead of a file. If the output is a single line, the result is a scalar string. If the output has multiple lines, the result is a string array with each element of the array containing a single line of the output.

### Example—Using STRING with Explicit Formatting

The IDL statements:

```

;Produce a string array.
A=STRING(FORMAT='("The values are:", /, (I))', INDGEN(5))

```

```
;Show its structure.  
HELP, A  
  
;Print out the result.  
FOR I = 0, 5 DO PRINT, A[I]
```

produce the following output:

```
A                STRING    = Array(6)  
The values are:  
  0  
  1  
  2  
  3  
  4
```

See [STRING](#) in the *IDL Reference Guide* for details.

## Reading Data from a String Variable

The READS procedure performs formatted input from a string variable and writes the results into one or more output variables. This procedure differs from the READ procedure only in that the input comes from memory instead of a file.

This routine is useful when you need to examine the format of a data file before reading the information it contains. Each line of the file can be read into a string using READF. Then the components of that line can be read into variables using READS.

See the description of READS in the *IDL Reference Guide* for more details.

# UNIX-Specific Information

UNIX offers only a single type of file. All files are considered to be an uninterpreted stream of bytes, and there is no such thing as record structure at the operating system level. (By convention, records of text are simply terminated by the linefeed character, which is referred to as “newline.”) It is possible to move the current file pointer to any arbitrary position in the file and to begin reading or writing data at that point. This simplicity and generality form a system in which any type of file can be manipulated easily using a small set of file operations.

## Reading FORTRAN-Generated Unformatted Data with IDL

The UNIX file system considers all files to be an uninterpreted stream of bytes. Standard FORTRAN I/O considers all input/output to be done in terms of logical records.

In order to reconcile the FORTRAN need for logical records with UNIX files, UNIX FORTRAN programs add a longword count before and after each logical record of data. These longwords contain an integer count giving the number of bytes in that record. Note that direct-access FORTRAN I/O does not write data in this format, but simply transfers binary data to or from the file.

The use of the `F77_UNFORMATTED` keyword with the `OPENR` statement informs IDL that the file contains unformatted data produced by a UNIX FORTRAN program. When a file is opened with this keyword, IDL interprets the longword counts properly and is able to read and write files that are compatible with FORTRAN.

### Reading data from a FORTRAN file

The following UNIX FORTRAN program produces a file containing a five-column by three-row array of floating-point values with each element set to its one-dimensional subscript:

```
PROGRAM ftn2idl

INTEGER i, j
REAL data(5, 3)

OPEN(1, FILE="ftn2idl.dat", FORM="unformatted")
DO 100 j = 1, 3
  DO 100 i = 1, 5
    data(i,j) = ((j - 1) * 5) + (i - 1)
    print *, data(i,j)
```

```

100 CONTINUE
    WRITE(1) data
END

```

Running this program creates the file *ftn2idl.dat* containing the unformatted array. The following IDL statements can be used to read this file and print out its contents:

```

;Create an array to contain the fortran array.
data = FLTARR(5,3)

;Open the fortran-generated file. The F77_UNFORMATTED keyword is
;necessary so that IDL will know that the file contains unformatted
;data produced by a UNIX FORTRAN program.
OPENR, lun, 'ftn2idl.dat', /GET_LUN, /F77_UNFORMATTED

;Read the data in a single input operation.
READU, lun, data

;Release the logical unit number and close the fortran file.
FREE_LUN, lun

;Print the result.
PRINT, data

```

Executing these IDL statements produces the following output:

```

0.00000    1.00000    2.00000    3.00000    4.00000
5.00000    6.00000    7.00000    8.00000    9.00000
10.0000    11.0000    12.0000    13.0000    14.0000

```

Because unformatted data produced by UNIX FORTRAN unformatted WRITE statements are interspersed with extra information before and after each logical record, it is important that the IDL program read the data in the same way that the FORTRAN program wrote it. For example, consider the following attempt to read the above data file one row at a time:

```

;Create an array to contain one row of the Fortran array.
data = FLTARR(5, /NOZERO)

OPENR, lun, 'ftn2idl.dat', /GET_LUN, /F77_UNFORMATTED

;One row at a time.
FOR I = 0, 4 DO BEGIN

;Read a row of data.
READU, lun, data

;Print the row.
PRINT, data

```

```

ENDFOR

;Close the file.
FREE_LUN, lun

```

Executing these IDL statements produces the output:

```

0.00000      1.00000      2.00000      3.00000      4.00000
% READU: End of file encountered. Unit: 100
           File: ftn2idl.dat6
% Execution halted at $MAIN$(0).

```

Here, IDL attempted to read the single logical record written by the FORTRAN program as if it were written in five separate records. IDL hit the end of the file after reading the first five values of the first record.

### Writing data to a FORTRAN file

The following IDL statements create a five-column by three-row array of floating-point values with each element set to its one-dimensional subscript, and writes the array to a data file suitable for reading by a FORTRAN program:

```

;Create the array.
data = FINDGEN(5,3)

;Open a file for writing. Note that the F77_UNFORMATTED keyword is
;necessary to tell IDL to write the data in a format readable by a
;FORTRAN program.
OPENW, lun, 'idl2ftn.dat', /GET_LUN, /F77_UNFORMATTED

;Write the data.
WRITEU, lun, data

;Close the file.
FREE_LUN, lun

```

The following FORTRAN program reads the data file created by IDL:

```

PROGRAM idl2ftn

INTEGER i, j
REAL data(5, 3)

OPEN(1, FILE="idl2ftn.dat", FORM="unformatted")
READ(1) data
DO 100 j = 1, 3
  DO 100 i = 1, 5
    PRINT *, data(i,j)
100 CONTINUE
END

```

# VMS-Specific Information

Input/output under VMS is a relatively complex topic, involving a large number of formats and options. VMS files are record-oriented, and it is necessary to take this into account when writing applications, especially those that will run under other operating systems. The VMS user faces decisions in the following areas:

## Organization

A VMS file can have sequential, relative, or indexed organization. The organization controls the way in which data is placed in the file and determines the options for random access. IDL is able to read data from all three organizations and is able to create sequential or indexed files.

In addition, it is possible to bypass the organization and access a file in “block mode.” In block mode, most VMS file processing is bypassed. The IDL user can access a block mode file as if it were simply a stream of uninterpreted bytes. This is very similar to stream files (although considerably more efficient).

## Warning

---

With some file organizations, VMS intermingles housekeeping information with data. When accessing such a file in block mode, it is easy to corrupt this information and render the file unusable in its usual mode; however, block mode will always work. Avoiding such corruption is the user’s responsibility.

---

## Access

The access mode controls how data in a file are accessed. VMS supports sequential access, random access by key value (indexed files), relative record number (relative files), or relative file address (all file organizations). IDL does not support access by relative record number—files are accessed sequentially or through key value. Random access for sequential files is allowed by file address using the `POINT_LUN` procedure.

## Record Format

VMS supports fixed-length records, variable-length records, variable length with fixed-length control field (VFC), and stream format. Of these, the fixed-length and variable-length record formats are the most useful and are fully supported by IDL.

It is possible to read the data portion of a VFC file, but not the control field. All access to stream mode files under IDL is done through the Standard C Library. It is worth noting that VMS stream files are record oriented (and therefore, fail to provide

much of the flexibility of UNIX stream files) although the VMS Standard C Library (upon which IDL is implemented) does a good job of concealing this limitation. Our experience indicates that input/output using VMS stream mode files is dramatically slower than the other options and should be avoided when possible. For unformatted data, using block mode can give similar flexibility as well as high efficiency.

## Record Attributes

When a record is output to the screen or printer, VMS uses its carriage control attributes to determine how to output each line. Explicit carriage control specifies that VMS should do nothing, and the user will provide the appropriate carriage control (if any) in the data. Carriage-return carriage control specifies that each line should be preceded by a line feed and followed by a carriage return. FORTRAN carriage control indicates that the first byte of each record contains a FORTRAN carriage control character. The possible values of this byte are given in the following table. The default for IDL is carriage-return carriage control.

Byte Value	ASCII Character	Meaning
0	(null)	No carriage control—output data directly.
32	(space)	Single-space. A linefeed precedes the output data, and a carriage return follows.
48	0	Double-space. Two linefeeds precede the output data, and a carriage return follows.
49	1	Page eject. A formfeed precedes the data, and a carriage return follows.
40	+	Overprint. A carriage return follows the data, causing the next output line to overwrite the current one.
36	\$	Prompt. A linefeed precedes the data, but no carriage return follows.
other		Same as ASCII space character. Single-space carriage control

Table 16-11: VMS FORTRAN Carriage Control

## File Attributes

There are many file attributes that can be adjusted to suit various requirements. These attributes allow specifying the default name, the initial size of new files, the amount

by which files are extended, whether the file is printed or sent to a batch queue when closed, file sharing between processes, etc.

## How IDL Handles Records

With record-oriented files, IDL always transfers at least a single record of data. If the amount of data required exceeds a single record, more input/output occurs. For example, consider the case of a file open on unit 1 for output with 80-character records. The statement,

```
WRITEU, UNIT, FINDGEN(512)
```

requires 2,048 bytes to be output (each floating-point value takes four bytes), and thus, causes 26 records to be output. The last record will not be entirely full and is padded at the end with zeroes.

On later input, the same rule is applied in reverse—26 records are read, and the unused portion of the last one is discarded. The basic rule of input/output with record-oriented files is that the form of the input and output statements should match. For instance, the statements,

```
WRITEU, UNIT, A
WRITEU, UNIT, B
WRITEU, UNIT, C
```

generate three output records and should be later input with statements of the following form:

```
READU, UNIT, A
READU, UNIT, B
READU, UNIT, C
```

In contrast, the statement

```
WRITEU, UNIT, A, B, C
```

generates a single-output record and should be later input with the following single statement:

```
READU, UNIT, A, B, C
```

## Reading FORTRAN-Generated Unformatted Data with IDL

The following VMS FORTRAN program produces a file containing a 5 x 5 array of floating-point values with each element set to its one-dimensional subscript:

```
INTEGER I, J REAL DATA(5, 5)
OPEN(1, FILE='data.dat', FORM='unformatted', status='new')
DO 100 J = 1, 5
```



```

        DO 100 I = 1, 5
            DATA(I,J) = ((J-1) * 5) + (I-1)
100 CONTINUE
        WRITE(1) DATA
    END

```

Running this program creates the file *data.dat* containing the unformatted data. By default, VMS FORTRAN programs create such files using *segmented records*, which is a scheme used by FORTRAN to write data records with lengths that exceed the actual record lengths allowed by VMS. Each segmented record is written as one or more actual VMS records. Each of the actual records has a 2-byte control field prepended that allows FORTRAN to reconstruct the original record. IDL is able to read and write segmented record files if the OPEN statement, used to access the file, includes the SEGMENTED keyword. The following IDL statements can be used to read this file and print out its contents:

```

;Open the file. The SEGMENTED keyword is necessary so that IDL will
;know that the file contains VMS FORTRAN segmented records.
OPENR, 1, 'data.dat', /SEGMENTED

;Create an array to contain the array.
A = FLTARR(5, 5, /NOZERO)

;Read the data in a single input operation.
READU, 1, A

;Print the result.
PRINT, A

```

Executing these IDL statements produces the following output:

```

0.00000 1.00000 2.00000 3.00000 4.00000
5.00000 6.00000 7.00000 8.00000 9.00000
10.0000 11.0000 12.0000 13.0000 14.0000
15.0000 16.0000 17.0000 18.0000 19.0000
20.0000 21.0000 22.0000 23.0000 24.0000

```

As with all record-oriented input/output, it is important that the IDL program read the data in the same way it was written by the FORTRAN program. For example, consider the following attempt to read the above data file one row at a time:

```

;Create an array to contain one row of the array.
OPENR, 1, 'DATA.DAT', /SEGMENTED
A = FLTARR(5, /NOZERO)

;One row at a time.
FOR I = 0, 4 DO BEGIN $

;Read a row of data.

```

```

READU, 1, A $

;Print the row.
PRINT, A $

ENDFOR

```

Executing these IDL statements produces the following output:

```

0.00000      1.00000      2.00000      3.00000      4.00000
% End of file encountered on file unit: 1.
% Execution halted at $MAIN$(0).

```

This program attempted to read the single logical record written by the FORTRAN program as if it were written in five separate records and so, hit the end of the file after reading the first five values of the first record.

## Indexed Files

### Creating Indexed Files

Although IDL can read and write indexed files, it cannot create them. The options for creating indexed files are so numerous that they should be specified using the VMS *CREATE/FDL* command. FDL (File Definition Language) is the standard method for specifying VMS file attributes. The *VAX/VMS File Definition Language Facility Reference Manual* (1986) describes FDL in detail. It is often useful to start with the FDL description for an existing file and then modify it to suit your new application. The VMS command,

```
$ ANALYZE/RMS FILE/FDL file.dat
```

creates a file named `file.fdl` containing the FDL description for `file.dat`. The following is an example of an FDL description for an indexed file named `data.dat` with two keys. The first key is a 32-character string containing an employee name. The second is a 4-byte integer containing the current salary for that employee:

```

FILE
  NAME data.dat
  ORGANIZATION indexed
RECORD
  SIZE 36
KEY 0
  NAME "Name"
  SEGO_LENGTH 32
  SEGO_POSITION 0
  TYPE string
KEY 1
  CHANGES yes

```

```

NAME "Salary"
SEG0_LENGTH 4
SEG0_POSITION 32
TYPE bin4

```

Assume that this description resides in a file named *data.fdl*. The following IDL statement can be used to create *data.dat*:

```
SPAWN, 'create/fdl = data.fdl'
```

Once the file exists, it can be opened within IDL using the KEYED keyword with the OPENR or OPENU procedures.

## Using Indexed Files

Given a file created using the FDL description in the previous section, the IDL statements below do four things:

- Add some employee records to the file
- Print the records out sorted by name
- Give an employee a raise
- Print the records sorted by increasing salary

IDL is able to perform both formatted and unformatted input/output with indexed files. In this instance, unformatted access is required because the record definition contains a binary field (salary).

```

;Open the previously created, empty file.
OPENU, UNIT, 'data.dat', /KEY, /GET_LUN

;Add the first record. The STRING function is used to pad the name
;to 32 characters using space characters because the data must
;match the FDL description of the file exactly.
WRITEU, UNIT, STRING('Natasha', FORMAT = "(A,T33)"), 14257L

;Second record.
WRITEU, UNIT, STRING('Bullwinkle', FORMAT = "(A,T33)"), 32501L

;Third record.
WRITEU, UNIT, STRING('Rocky', FORMAT = "(A,T33)"), 32500L

;Fourth and last record.
WRITEU, UNIT, STRING('Borris', FORMAT = "(A,T33)"), 6805L

;Print the contents of the file, sorted by name. READ_BY_INDEX is a
;procedure (described below) that does the actual work.
READ_BY_INDEX, UNIT, 0, 'a', 'By Name:'

```

```

;In preparation for giving a raise, make variables to read the
;current information on the employee.
NAME = STRING(REPLICATE(32B, 32))

SALARY = 0L

;Read the record for employee Bullwinkle.
READU, UNIT, NAME, SALARY, KEY_VALUE = 'Bullwinkle'

;Update Bullwinkle's record with an increased salary. The REWRITE
;keyword causes the last input record to be overwritten, instead of
;creating a new record.
WRITEU, UNIT, NAME, SALARY + 10000L, /REWRITE

;Print the contents of the file, sorted by salary.
READ_BY_INDEX, UNIT, 1, 0L, 'By Salary:'

;Free the file unit, and close the file.
FREE_LUN, UNIT

```

The procedure `READ_BY_INDEX` is implemented as follows:

```

;Print the contents of the file sorted on the index given by KI. KV
;is the value the first record should be matched against. Heading
;is a banner comment to be printed before the file contents.
PRO READ_BY_INDEX, UNIT, KI, KV, HEADING

;Indicates first trip through main loop.
FIRST = 1

;Prepare variables to read the records into.
NAME = STRING(REPLICATE(32B, 32))
SALARY = 0L

;The EOF function does not work with indexed files, so we will use
;ON_IOERROR to catch attempts to read too far.
ON_IOERROR, EOD

;Loop will be exited on end-of-file.
WHILE 1 DO BEGIN

    ;First iteration.
    IF (FIRST) THEN BEGIN

        ;Output the heading.
        PRINT, FORMAT='(/, a)', HEADING)

        ;On first iteration, use keywords to locate the first record.

```

```

        READU, UNIT, NAME, SALARY, KEY_ID = KI, KEY_MATCH = 1, $
            KEY_VALUE = KV

;Indicate that first iteration has happened.
        FIRST = 0

;After the first iteration, use normal input statement to read
;sequentially.
        ENDIF ELSE BEGIN

            READU, UNIT, NAME, SALARY

        ENDELSE

        ;Print the record.
        PRINT, FORMAT = '(4X, A, T15, I)', NAME, SALARY

    ENDWHILE

;When the above loop tries to read past end-of-file, execution will
;be transferred here.
EOD:

END

```

Executing the above statements gives the following output:

```

By Name:
    Borris                6805
    Bullwinkle            32501
    Natasha               14257
    Rocky                 32500
By Salary:
    Borris                6805
    Natasha               14257
    Rocky                 32500
    Bullwinkle            42501

```

## Magnetic Tape

Under VMS, IDL offers procedures to directly access magnetic tapes. Data are transferred between the tape and IDL arrays without using RMS. Optionally, tapes from IBM mainframe compatible systems may be read or written with odd/even byte reversal.

The routines used to access magnetic tape directly are as follows:

Routine	Description
REWIND	Rewind a tape unit.
SKIPF	Skip records or files.
TAPRD	Read from tape.
TAPWRT	Write to tape.
WEOF	Write an end-of-file mark on tape.

*Table 16-12: Magnetic Tape Access Routines*

To use the IDL magnetic tape procedures, you must define a logical name *MTn*: to be equivalent to the actual name of the tape drive you wish to use. This definition must be done before invoking IDL. You also must have the tape mounted as a foreign volume.

For example, if you wish to access the tape drive MUA0: as IDL tape unit number one, issue the following VMS commands before running IDL:

```
$ MOUNT/FOREIGN MUA0:
$ DEFINE MT1 MUA0:
```

Then, within IDL, refer to the tape as unit number one. The IDL unit number *n* may range from 0 to 9.

### Note

These unit numbers are not the same as the LUNs used by the other input/output routines. The unit numbers used by the magnetic tape routines are completely unrelated and come from the last letter of the MT\* logical name used to refer to it.

## Magnetic Tape Examples

The following statements skip forward 30 records on the tape mounted on the drive with the logical name MT2: and print a message if an end-of-file was encountered.

```
;Skip forward over 30 records on unit 2.
SKIPF, 2, 30, 1

;Print a message if the requested number of records were not
;skipped.
IF !ERR NE 30 THEN PRINT, 'end-of-file hit'
```

The next example skips two files backwards and then positions the tape immediately after the second file mark encountered in reverse.

```

;Go backwards two files. Position after file if two files were
;actually skipped.
SKIPF, 0, -2

IF !ERR EQ -2 THEN SKIPF, 0, 1

```

The following code segment reads a 512 x 512-byte image from the tape which is assigned the logical name MT5. It is assumed that the data are written in 2,048-byte tape blocks.

```

;Define image array.
a = BYTARR(512, 512)

;Define an array to hold one tape block worth of data.
b = BYTARR(512, 4)

FOR I = 0, 511, 4 DO BEGIN
  ;Read next record.
  TAPRD, B, 5

  ;Insert four rows starting at i-th row.
  A[0, I] = B

ENDFOR

```

Assuming the tape is actually on drive MXB2:, the mount command, which must be issued to VMS before entering IDL, is as follows:

```

;This command serves to both mount the tape and define the logical
;name MT5 to refer to it, thus making it unit 5 within IDL.
$ MOUNT MXB2:/FOR "" MT5

```

## References

Digital Equipment Corporation (1986), *VAX/VMS File Definition Language Facility Reference Manual*, Order Number AA-Z415B-TE, Maynard, Massachusetts.

## Windows-Specific Information

Under Microsoft Windows, a file is read or written as an uninterrupted stream of bytes—there is no record structure at the operating system level. Files are processed as *binary* or *text*. *Binary* files are processed using no translation of characters. *Text* files are processed by translating the characters that terminate a line. Lines are terminated by the character sequence CR LF (carriage return, line feed). During read operations, if a CR character precedes a LF character, the CR is removed. During write operations, all LF characters are prepended with a CR character.

The ASSOC, READU, and WRITEU routines operate in binary mode by default. The PRINT, PRINTF, READ, and READF routines operate in text mode by default. You can override the defaults by setting the BINARY or NOAUTOMODE keywords to the OPEN procedures. See the documentation for the Windows-Only keywords to [OPEN](#) in the *IDL Reference Guide*.



## Macintosh-Specific Information

Macintosh files store two pieces of information not generally stored by files on other platforms—the file's *type* and its *creator*. The `MACTYPE` and `MACCREATOR` keywords to the `OPEN` procedures allow you to explicitly set the type and creator for files created on a Macintosh. See the documentation for the Macintosh-Only keywords to `OPEN` in the *IDL Reference Guide*.

## Scientific Data Formats

IDL supports the HDF (Hierarchical Data Format), HDF-EOS (Hierarchical Data Format-Earth Observing System), CDF (Common Data Format), and NetCDF (Network Common Data Format) self-describing, scientific data formats. Collections of built-in routines provide an interface between IDL and these formats. Documentation for specific routines and further discussion of the various formats can be found in *IDL Scientific Data Formats Guide*.

## Support for Standard Image File Formats

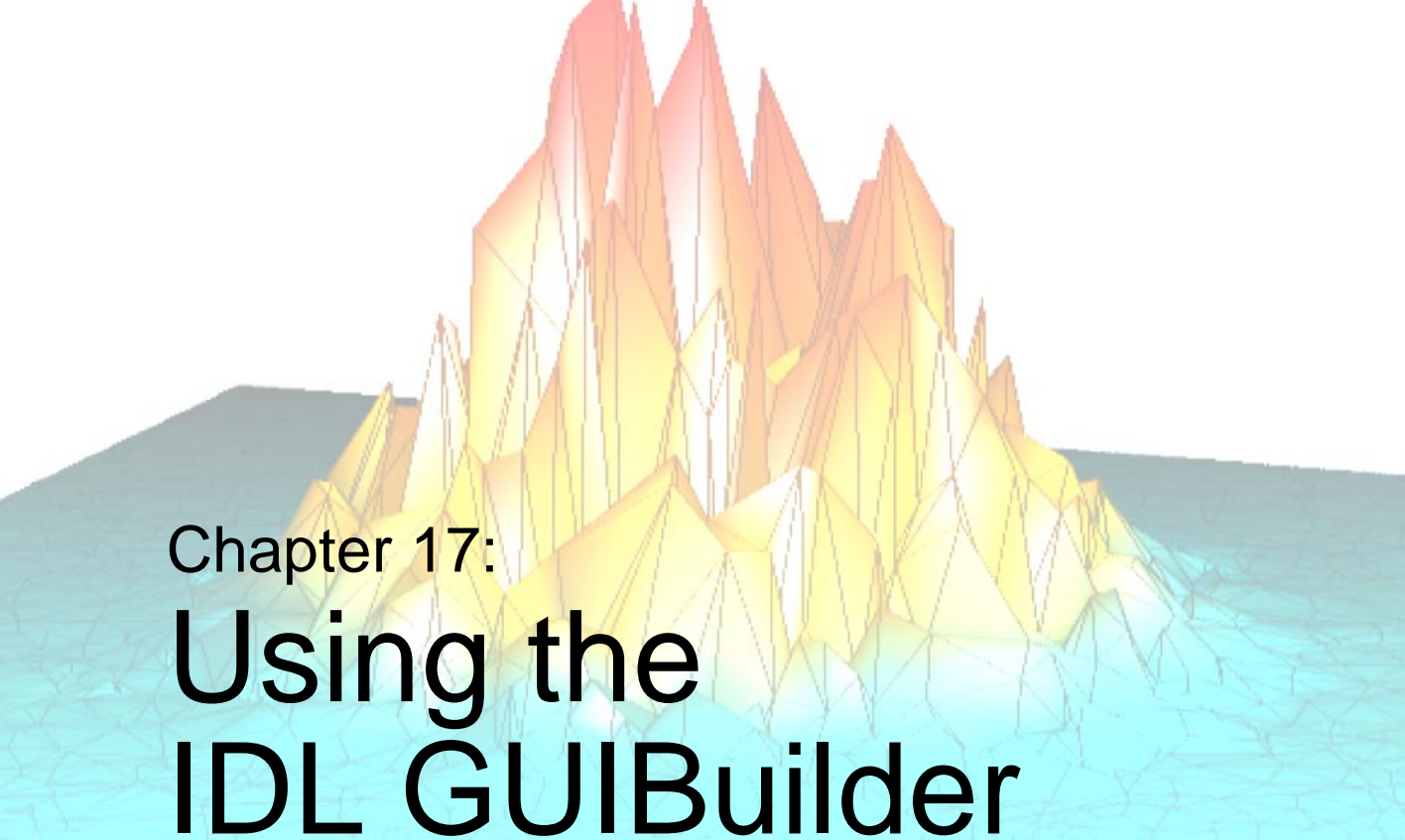
IDL includes routines for reading and writing many standard graphics file formats. These routines and the types of files they support are listed in the table below. Documentation on these routines can be found in the online help (enter “?” at the IDL prompt).

<b>Format</b>	<b>Read/Write Routines</b>	<b>Query Routine</b>	<b>Description</b>
BMP	<a href="#">READ_BMP</a> <a href="#">WRITE_BMP</a>	<a href="#">QUERY_BMP</a>	Windows Bitmap (.bmp) Format
GIF	<a href="#">READ_GIF</a> <a href="#">WRITE_GIF</a>	<a href="#">QUERY_GIF</a>	CompuServe Graphics Interchange Format
Interfile	<a href="#">READ_INTERFILE</a> (Write routine is n/a)	n/a	Interfile version 3.3 Format
JPEG	<a href="#">READ_JPEG</a> <a href="#">WRITE_JPEG</a>	<a href="#">QUERY_JPEG</a>	Joint Photographic Experts Group files
NRIF	(Read routine is n/a) <a href="#">WRITE_NRIF</a>	n/a	NCAR Raster Interchange Format
PICT	<a href="#">READ_PICT</a> <a href="#">WRITE_PICT</a>	<a href="#">QUERY_PICT</a>	Macintosh version 2 PICT files (bitmap only)
PNG	<a href="#">READ_PNG</a> <a href="#">WRITE_PNG</a>	<a href="#">QUERY_PNG</a>	Portable Network Graphics file
PPM	<a href="#">READ_PPM</a> <a href="#">WRITE_PPM</a>	<a href="#">QUERY_PPM</a>	PPM/PGM Format
SRF	<a href="#">READ_SRF</a> <a href="#">WRITE_SRF</a>	<a href="#">QUERY_SRF</a>	Sun Raster File
TIFF	<a href="#">READ_TIFF</a> <a href="#">WRITE_TIFF</a>	<a href="#">QUERY_TIFF</a>	8-bit or 24-bit Tagged Image File Format
X11 Bitmap	<a href="#">READ_X11_BITMAP</a> (Write routine is n/a)	n/a	X11 Bitmap format used for reading bitmaps for IDL widget button labels

*Table 16-13: IDL-Supported Graphics Standards*

<b>Format</b>	<b>Read/Write Routines</b>	<b>Query Routine</b>	<b>Description</b>
XWD	<a href="#">READ_XWD</a> (Write routine is n/a)	n/a	X Windows Dump format

*Table 16-13: IDL-Supported Graphics Standards*



# Chapter 17: Using the IDL GUIBuilder

The following topics are covered in this chapter:

---

Overview .....	438	Base Widget Properties .....	496
Starting the IDL GUIBuilder .....	440	Button Widget Properties .....	507
Creating an Example Application .....	442	Text Widget Properties .....	511
IDL GUIBuilder Tools .....	453	Label Widget Properties .....	516
Widget Operations .....	468	Slider Widget Properties .....	518
Generating Files .....	471	Droplist Widget Properties .....	521
IDL GUIBuilder Examples .....	473	Listbox Widget Properties .....	523
Widget Properties .....	489	Draw Widget Properties .....	526
Common Widget Properties .....	490	Table Widget Properties .....	532

# Overview

The IDL GUIBuilder is part of the IDLDE for Windows. The IDL GUIBuilder supplies you with a way to interactively create user interfaces and then generate the IDL source code that defines that interface and contains the event-handling routine place holders.

---

**Note**

The IDL GUIBuilder is supported on Windows only. However, the code it generates is portable and runs on all IDL supported platforms. Since applications built with IDL GUIBuilder may require functionality added in the current release, generated code only runs on the version of IDL you generated the code on or greater.

---

The IDL GUIBuilder has several tools that simplify application development. These tools allow you to create the widgets that make up user interfaces, define the behavior of those widgets, define menus, and create and edit color bitmaps for use in buttons.

---

**Note**

When using code generated by the IDL GUIBuilder on other non-Windows platforms, more consistent results are obtained by using a row or column layout for your bases instead of a bulletin board layout. By using a row or column layout, problems caused by differences in the default spacing and decorations (e.g., beveling) of widgets on each platform can be avoided

---

These are the basic steps you will follow when building an application interface using the IDL GUIBuilder:

1. Interactively design and create a user interface using the components, or *widgets*, supplied in the IDL GUIBuilder. Widgets are simple graphical objects supported by IDL, such as sliders or buttons.
2. Set attribute properties for each widget. The attributes control the display, initial state, and behavior of the widget.
3. Set event properties for each widget. Each widget has a set of events to which it can respond. When you design and create an application, it is up to you to decide if and how a widget will respond to the events it can generate. The first step to having a widget respond to an event is to supply an event procedure name for that event.

4. Save the interface design to an IDL resource file, \*.prc file, and generate the portable IDL source code files. There are two types of generated IDL source code: widget definition code (\*.pro files) and event-handling code (\*\_eventcb.pro files).
5. Modify the generated \*\_eventcb.pro event-handling code file using the IDLDE, then compile and run the code. This code can run on any IDL-supported platform.

The \*\_eventcb.pro file contains place holders for all of the event procedures you defined for the widgets, and you complete the file by filling in the necessary event callback routines for each procedure.

### Warning

---

Once you have generated the widget definition code (\*.pro files), you should not modify this file manually. If you decide to change your interface definition, you will need to regenerate the interface code, and will therefore overwrite that \*.pro file. Any new event handling code will not be overwritten but will instead be appended.

---

For information about IDL widgets, and how to create user interfaces programmatically (without the IDL GUIBuilder), see [Chapter 18, “Widgets”](#).

# Starting the IDL GUIBuilder

To open a new IDL GUIBuilder window:

From the IDLDE File menu, choose New, then choose GUI.

*Or*

Click the New GUI button on the IDLDE toolbar.

Each of these actions opens a new IDL GUIBuilder window and displays the IDL GUIBuilder toolbar. The IDL GUIBuilder window contains a top-level base widget, as shown in the following figure. This top-level base holds all of the widgets for an individual interface; it is the top-level parent in the widget hierarchy being created.

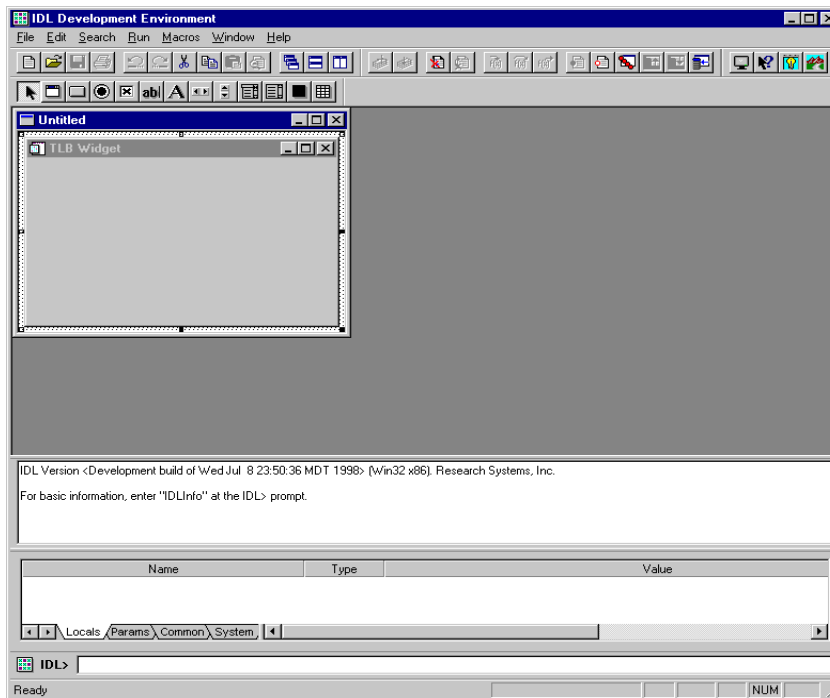


Figure 17-1: IDLDE with IDL GUIBuilder Window

## Opening Existing Interface Definitions

To open an existing interface design in the IDL GUIBuilder:



1. Do one of the following to launch the Open dialog:

From the IDLDE File menu, choose Open.

*Or*

Click on the Open button on the IDLDE toolbar.

2. In the Open dialog, locate and select the appropriate \*.prc file, and click Open.

The \*.prc portable resource file contains the widget definitions that make up the widget hierarchy and define your interface design. When you click Open, the existing definition is displayed in a IDL GUIBuilder window. You can modify the interface then save it, and you can generate new IDL source code for the modified definition.

# Creating an Example Application

The following example takes you through the process of creating your first application with the IDL GUIBuilder and the IDLDE. You will create the user interface and write the event callback routines.

The simple example application contains a menu and a draw widget. When complete, the running application allows the user to open and display a graphics file in GIF format, change the color table for the image display, and perform smooth operations on the displayed image.

This example introduces you to some of the basic procedures you will use to create applications with the IDL GUIBuilder; it shows you how to define menus, create widgets, set widget properties, and write IDL code to handle events.

## Defining Menus for the Top-level Base

To define the menu, follow these steps:

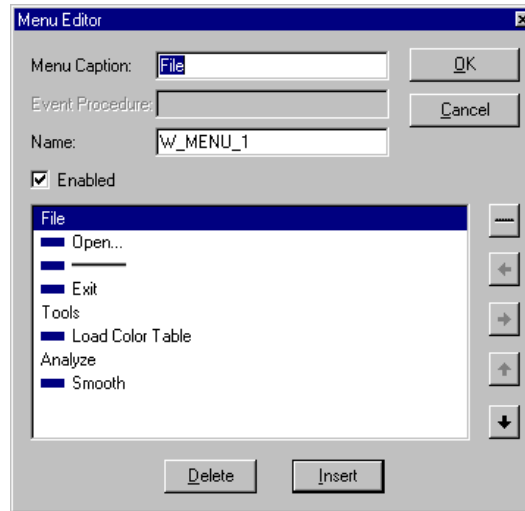
1. Open a new IDL GUIBuilder window by clicking on the New GUI button (window icon) on the IDLDE toolbar.
2. Drag out the window then the top-level base to a reasonable size for displaying an image.

For example, drag the base out so that it has an X Size property value of 500 and a Y Size property value of 400. To view the property values, right-click on the base, and choose Properties from menu. In the Properties dialog, scroll down to view the X Size and Y Size property values.

3. Right-click on the top-level base in the IDL GUIBuilder window, then choose Edit Menu. This action opens the Menu Editor.
4. In the Menu Editor Menu Caption field, enter “File” and click Insert. Clicking Insert sets the entered value and adds a new line after the currently selected line, and the new line becomes the selected line.
5. To define the File menu items, do the following:
  - A. With the new line selected, click on the right arrow in the Menu Editor, which indents the line and makes it a menu item.
  - B. Click in the Menu Caption field and enter “Open...”.
  - C. Click in the Event Procedure field and enter “OpenFile”. The OpenFile routine will be called when the user selects this menu.

- D. To create a separator after the Open menu, click the line button at the right side of the dialog (above the arrow buttons).
  - E. To set the values and move to a new line, click Insert.
  - F. In the Menu Caption field, enter “Exit”.
  - G. In the Event Procedure field, enter “OnExit”.
  - H. To set the values and move to a new line, click Insert.
6. To define the Tools menu and its one item, do the following:
    - A. With the new line selected, click the left arrow to make the line a top-level menu.
    - B. In the Menu Caption field, enter “Tools”, then click Insert.
    - C. Click the right arrow to make the new line a menu item.
    - D. In the Menu Caption field, enter “Load Color Table”.
    - E. In the Event Procedure field, enter “OnColor”.
    - F. To set the values and move to a new line, click Insert.
  7. To define the Analyze menu and its one menu item, do the following:
    - A. With the new line selected, click the left arrow to make the line a top-level menu.
    - B. In the Menu Caption field, type “Analyze”, then press Enter.
    - C. Click the right arrow to make the new line a menu item.
    - D. In the Menu Caption field, enter “Smooth”.
    - E. In the Event Procedure field, enter “DoSmooth”.

Your entries should look like those shown in the following figure.



*Figure 17-2: Menu Editor Dialog with Example Menus*

8. Save your menu definitions by clicking OK in the Menu Editor.

---

**Note**

For more information about using the Menu Editor, see [“Using the Menu Editor”](#) on page 461.

---

9. At this time you can click on the menus to test them. Your interface should look similar to the one in the following figure.
10. From the IDLDE File menu, choose Save, which opens the Save As dialog.
11. In the Save As dialog, select a location, enter “example.prc” in the File name field, and click Save. This action writes the portable resource code to the specified file.

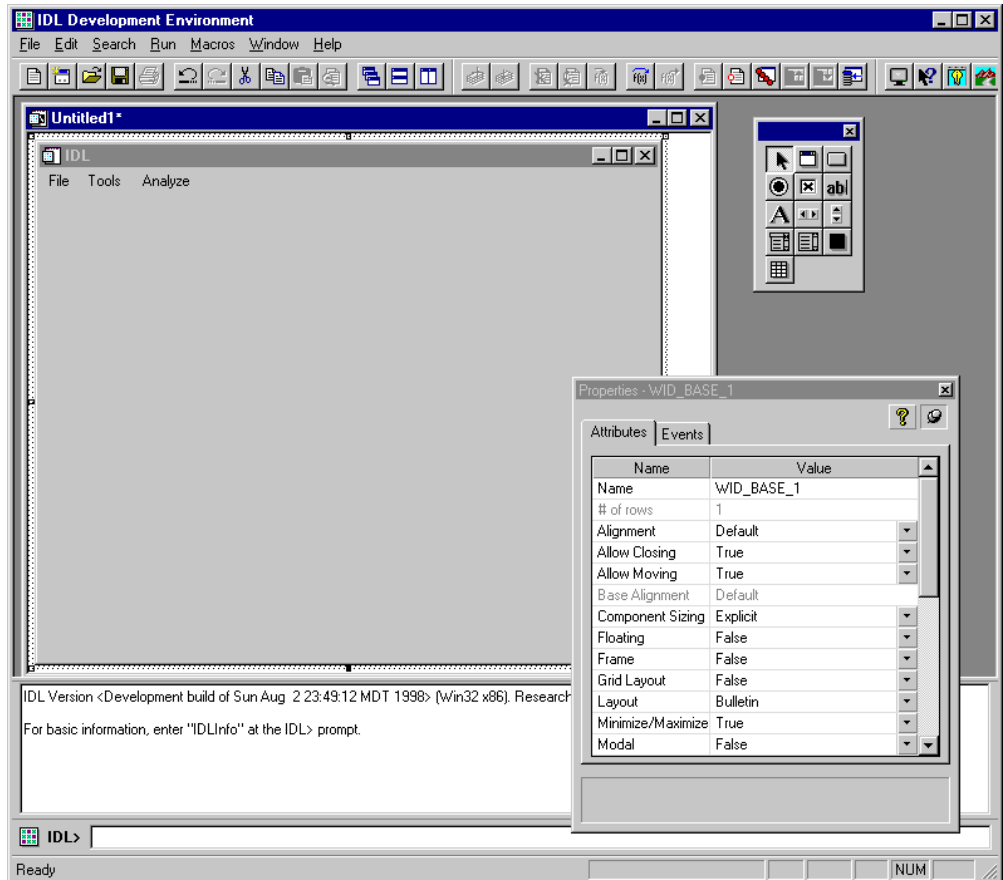


Figure 17-3: IDL GUIBuilder with Example Application

## Creating a Draw Widget

To create a draw area that will display GIF image files, follow these steps:

1. Click on the Draw Widget tool button (the dark square icon), then drag out an area that fills the top-level base display area. Leave a small margin around the edge of the draw area when you drag it out.

For more information about creating and operating on widgets, see [“Using the IDL GUIBuilder Toolbar”](#) on page 454 and [“Widget Operations”](#) on page 468.

2. Right click on the draw area, and choose Properties. This action opens the Properties dialog for the draw area; the draw widget properties are displayed in the dialog.
3. In the Properties dialog, click the push pin button so the dialog will stay open and on top.
4. In the Properties dialog, change the draw widget **Name** attribute value to “Draw”.

Later, you will write code to handle the display of the image in this draw area widget. Renaming the widget now will make it easier to write the code later; the “Draw” name is easy to remember and to type.

---

**Note**

The Name property must be unique to the widget hierarchy.

---

5. In the IDL GUIBuilder window, click on the top-level base widget to select it. When you do so, the Properties dialog will update and display the attributes for this base widget.
6. In the Properties dialog, locate the **Component Sizing** property, and select Default from the droplist values. This action sizes the base to the draw widget size you created.

When you first dragged out the size of the base, the Component Sizing property changed from Default to Explicit—you explicitly sized the widget. Now that the base widget contains items, you can return it to Default sizing, and IDL will handle the sizing of this top-level base.

7. From the File menu, choose Save, which saves your new modifications to the example.prc file. The application should look like the one shown in the following figure.

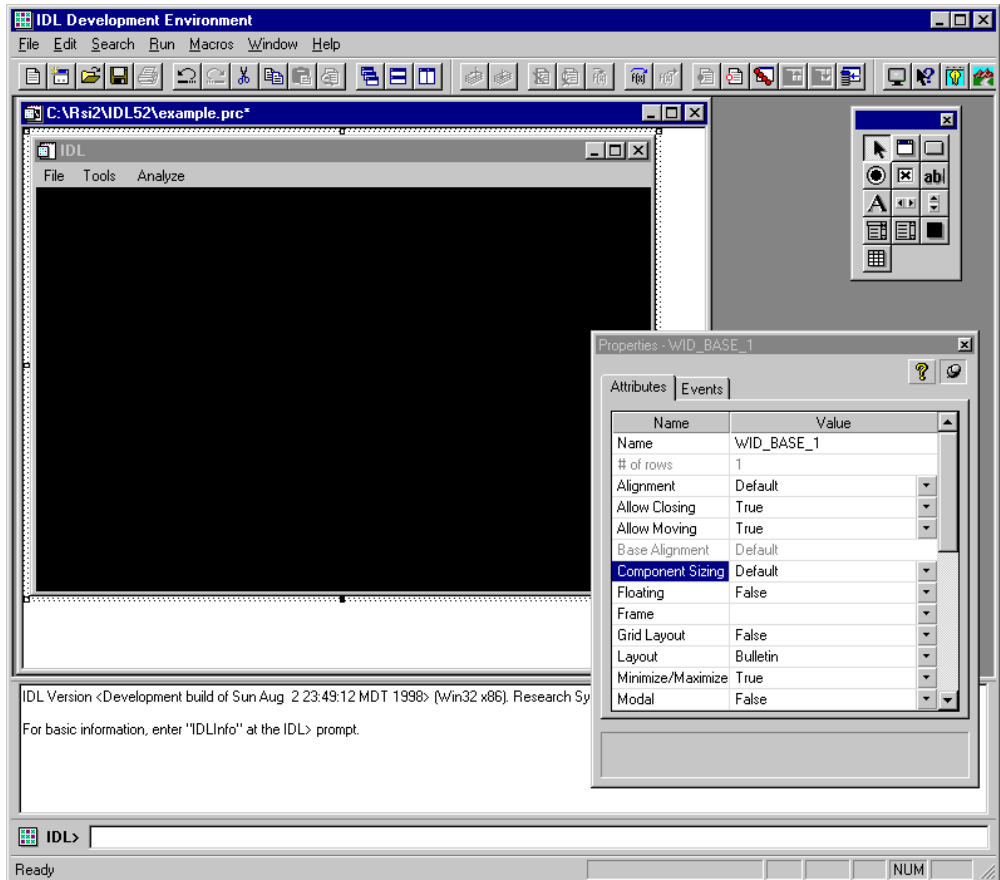


Table 17-1: Complete Example Application

## Running the Application in Test Mode

You can run the application in test mode, which allows you to test the display of widgets and menus.

To run your application in test mode:

From the Run menu, choose Test GUI.

*Or*

Press Control+t.

Both these actions display the interface as it will look when it runs. You can click on the menus, but there is no active event handling in test mode.

To exit test mode:

Press the Esc key.

*Or*

Click the close X in the upper-right corner of the test application window.

## Generating the IDL Code

To generate the code for the example application, follow these steps:

1. From the File menu, choose Generate .pro. This action opens the Save As dialog.
2. In the Save As dialog, find the location where you want the files saved, enter “example.pro” in the File name field, and click Save.

This action generates an example.pro widget definition file and an example\_eventcb.pro event-handling file.

The example.pro file contains the widget definition code, and you should never modify this file. If you decide later to change your interface, you will need to regenerate this interface code, and thus overwrite the widget code file.

The example\_eventcb.pro contains place holders for all the event procedures you defined in the IDL GUIBuilder Menu Editor and Properties dialog. You must complete these event procedures by filling in event callback routines. If you generate code after you have modified this file, any new event handling code will not be overwritten but will instead be appended. For information on ways to handle regenerating the \*\_eventcb.pro file, see [“Notes on Generating Code a Second Time”](#) on page 472.

For more information on interface definitions and generated code, see [“Generating Files”](#) on page 471.

---

### Note

You should modify *only* the generated event-handling file (\*\_eventcb.pro); you should never modify the generated interface code (the \*.pro file).

---



## Handling the Open File Event

You can now modify the generated `example_eventcb.pro` file to handle the events for the application. First, you will modify the `OpenFile` routine.

When the user selects `Open` from the `File` menu of the example application, the appropriate event structure is sent, and the `OpenFile` routine handles the event. For this application, the `Open` menu item will launch an `Open` dialog to allow the user to choose a `GIF` file, and then the routine will check the selected file's type, read the image, and display it in the draw area.

To open the file and add the code to handle the `OpenFile` event, follow these steps:

1. From the `File` menu in the `IDLDE`, choose `Open`, which launches the `Open` dialog.
2. In the `Open` dialog, locate and select the `example_eventcb.pro` file, and click `Open`. This file contains the event handling routine place holders, which you will now complete.
3. In the `example_eventcb.pro` file, locate the `OpenFile` routine calling sequence, which looks like this:

```
PRO OpenFile, Event
```

```
END
```

4. Add the following code to handle the event (the comments describe the added code):

```
PRO OpenFile, Event
```

```

; If there is a file, draw it to the draw widget.
sFile = Dialog_Pickfile(filter="*.gif")
if(sFile ne "")then begin

    ; Find the draw widget, which is named Draw.
    wDraw = Widget_Info(Event.top, find_by_uname="Draw");
    ; Make sure something was found.
    if(wDraw gt 0)then begin
        ; Make the draw widget the current, active window.
        widget_control, wDraw, get_value=idDraw
        wset,idDraw

        ; Read in the image.
        read_gif,sFile, im
        ; Size the image to fill the draw area.
        im = congrid(im, !d.x_size, !d.y_size)

```

```

        ; Display the image.
        tv, im
        ; Save the image in the uvalue of the top-level base.
        widget_control, Event.top, set_uvalue=im, /no_copy
    endif
endif
END

```

---

**Note**

In the added code, you used the `FIND_BY_UNAME` keyword to find the draw widget using its name property. In this example, the widget name, “Draw”, is the one you gave the widget in the IDL GUIBuilder Properties dialog. The widget name is case-sensitive.

---

## Handling the Exit Event

To add the code that causes the example application to close when the user chooses Exit from the File menu, follow these steps:

1. Locate the `OnExit` routine place holder, which looks like this:

```

PRO OnExit, Event

END

```

2. Add the following statement to handle the destruction of the application:

```

PRO OnExit, Event

    widget_control, Event.top, /destroy

END

```

## Handling the Load Color Table Event

To add the code that causes the example application to open the IDL color table dialog when the user chooses Load Color Table from the Tools menu, follow these steps:

1. Locate the `OnColor` routine place holder, which looks like this:

```

PRO OnColor, Event

END

```

2. Add the following procedure to open the IDL `XLoadct` color table dialog:

```

PRO OnColor, Event

```

```

        xloadct

    END

```

This procedure opens a dialog from which the user can select from a set of predefined color tables. When the user clicks the name of a color table, it is loaded and the displayed GIF file changes appropriately.

### Note

---

The IDL XLoadct color table dialog affects only 8-bit display devices.

---

## Handling the Smooth Event

When the user selects Smooth from the Analyze menu, a smooth operation is performed on the displayed GIF image. The smooth operation displays a smoothed image with a boxcar average of the specified width, which in the example code is 5.

To add the callback routines to handle the smooth operation, follow these steps:

1. Locate the DoSmooth routine place holder, which looks like this:

```

PRO DoSmooth, Event

    END

```

2. Add the following code to handle the smooth operation:

```

PRO DoSmooth, Event

    ; Get the image stored in the uvalue of the top-level-base.
    widget_control, Event.top, get_uvalue=image, /no_copy
    ; Make sure the image exists.
    if(n_elements(image) gt 0)then begin

        ; Smooth the image.
        image = smooth(image, 5)
        ; Display the smoothed image.
        tv, image
        ; Place the new image in the uvalue of the button widget.
        widget_control, Event.top, set_uvalue=image, /no_copy
    endif

    END

```

3. From the File menu, choose Save, which saves all your changes to the example\_eventcb.pro file.

## Compiling and Running the Example Application

To compile and run your example application, follow these steps:

1. At the IDL> command prompt, type the following:

```
example
```

2. This action compiles and runs the example application. The following figure shows the example application and the IDL color table dialog.

In the running application, you can open and display a GIF file. Then, you can open the IDL XLoadct dialog and change the color table used in displaying the image, or you can perform the smooth procedure on the image.

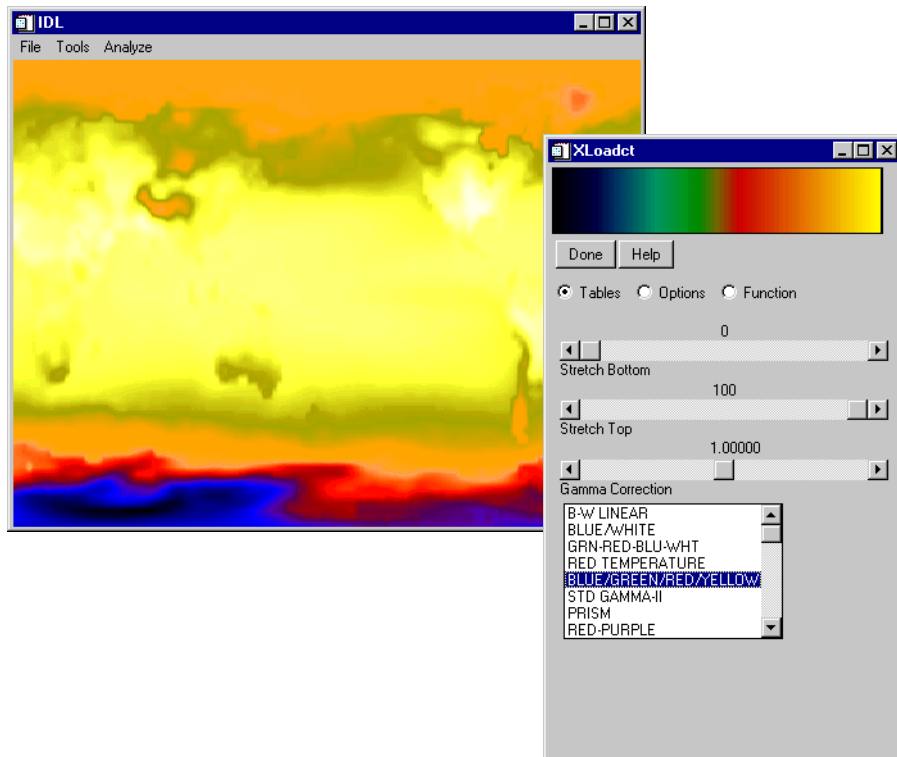


Table 17-2: Running Example Application and XLoadct Dialog

# IDL GUIBuilder Tools

You will use the following tools to design and construct a graphical interface using the IDL GUIBuilder:

- The IDL GUIBuilder Toolbar, which you use to create the widgets that make up your interface. See [“Using the IDL GUIBuilder Toolbar”](#) on page 454 and [“Widget Operations”](#) on page 468.
- Widget Properties dialog, which you use to set widget attributes and event properties. See [“Using the Properties Dialog”](#) on page 457 and [“Widget Properties”](#) on page 489.
- Widget Browser, which you can use to see the widget hierarchy and to modify certain aspects of the widgets in your application. See [“Using the Widget Browser”](#) on page 460.
- The Menu Editor, which you use to define menus to top-level bases and buttons. See [“Using the Menu Editor”](#) on page 461.
- The Bitmap Editor, which you use to create or modify bitmap images to be displayed on button widgets. See [“Using the Bitmap Editor”](#) on page 465.
- The IDLDE to modify, compile, and run the generated code (see [Chapter 3](#), [“The IDL for Windows Interface”](#) in the *Using IDL* manual).

## Using the IDL GUIBuilder Toolbar

The IDL GUIBuilder has its own toolbar in the IDE, which you use to create the widgets for your user interface. The following figure shows the toolbar.

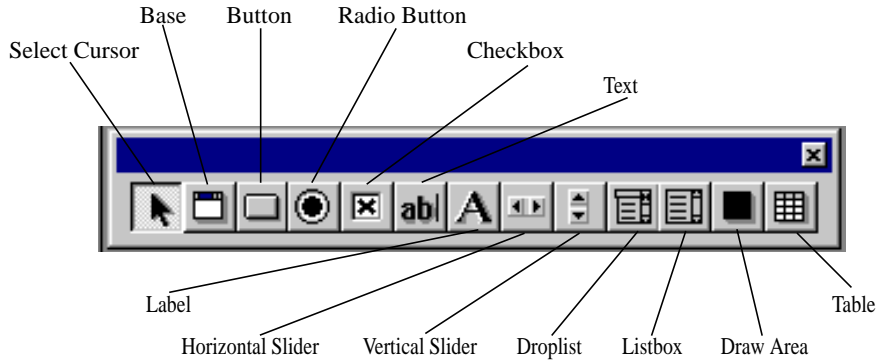


Figure 17-4: IDL GUIBuilder Toolbar

These are the widget types you can create using the IDL GUIBuilder toolbar:

Widget	Description
<b>Base</b>	Creates a container for a group of widgets within a top-level base container. A top-level base is contained in the IDL GUIBuilder window, and you build your interface in it. Use base widgets within the top-level base to set up the widget hierarchy, layout, and to organize the application. For example, you can use a base widget to group a set of buttons. For information on base properties, see <a href="#">“Base Widget Properties”</a> on page 496.
<b>Button</b>	Creates a push button. The easiest way to allow a user to interact with your application is through a button click. You can have button widgets display labels, menus, or bitmaps. For information on button properties, see <a href="#">“Button Widget Properties”</a> on page 507.

Table 17-3: Widget Types

Widget	Description
<b>Radio Button</b>	Creates a toggle button that is always grouped within a base container. Use radio buttons to present a set of choices from which the user can pick only one. For information on radio button properties, see <a href="#">“Button Widget Properties”</a> on page 507.
<b>Checkbox</b>	Creates a checkbox, which you can use either as a single toggle button to indicate a particular state is on or off or as a list of choices from which the user can select none to all choices. Checkboxes are created within a base container. For information on checkbox properties, see <a href="#">“Button Widget Properties”</a> on page 507.
<b>Text</b>	Creates a text widget. Use text widgets to get input from users or to display multiple lines of text. For information on text widget properties, see <a href="#">“Text Widget Properties”</a> on page 511.
<b>Label</b>	Creates a label. Use label widgets to identify areas of your application or to label widgets that do not have their own label property. Use labels when you have only a single line of text and you do not want the user to be able to change the text. For information on label widget properties, see <a href="#">“Label Widget Properties”</a> on page 516.
<b>Horizontal and Vertical Sliders</b>	Creates a slider with a horizontal or vertical layout. Use slider widgets to allow the user to control program input, such as adjust the speed of movement for a rotating image. For information on slider properties, see <a href="#">“Slider Widget Properties”</a> on page 518.
<b>Droplist</b>	Creates a droplist widget, which you can use to present a scrollable list of items for the user to select from. The droplist is an effective way to present a lot of choices without using too much interface space. For information on droplist properties, see <a href="#">“Droplist Widget Properties”</a> on page 521.
<b>Listbox</b>	Creates a list widget, which you can use to present a scrollable list of items for the user to select from. For information on listbox properties, see <a href="#">“Listbox Widget Properties”</a> on page 523.

Table 17-3: Widget Types

Widget	Description
<b>Draw Area</b>	Creates a draw area, which you can use to display graphics in your application. The draw area can display IDL Direct Graphics or IDL Object Graphics, depending on how you set its properties. For information on the draw area properties, see <a href="#">“Draw Widget Properties”</a> on page 526.
<b>Table</b>	Creates a table widget, which you can use to display data in a row and column format. You can allow users to edit the contents of the table. For information on the table widget properties, see <a href="#">“Table Widget Properties”</a> on page 532.

*Table 17-3: Widget Types*

### Note

The Select Cursor button returns the cursor to its standard state, and it indicates that the cursor is in that state. After you click on another button and create the selected widget, the cursor returns to the selection state.

## Creating Widgets

All widgets for a user interface must be descendents of a top-level base; in the IDL GUIBuilder window, all widgets must be contained in a top-level base widget. When you open a IDL GUIBuilder window, it contains a top-level base. You can add base widgets to that top-level widget to form a widget hierarchy. The added bases can act as containers for groups of widgets.

To create a widget:

Click on the appropriate button on the toolbar, then drag out an area within the top-level base widget. When you release the mouse button, a widget the size of the dragged-out area is created.

*Or*

Click on the appropriate button on the toolbar, then click within the top-level base area. This action creates a widget of the default size.

After you add widgets to a top-level base, you can resize, move, and delete them, and you can change their parent base. You can also set properties for each widget. For information on how to operate on widgets, see [“Widget Operations”](#) on page 468, and for information on setting properties, see [“Using the Properties Dialog”](#) on page 457.



## Using the Properties Dialog

For each widget, you can define attribute and event procedure properties. A widget's attributes define how it will display on the screen and its basic behaviors. The attributes you can set for a selected widget are displayed on the Attributes tab of the Properties dialog. These attributes are initially set to default values.

Event procedures are the predefined set of events a widget can recognize. When you write an application, you decide if and how the widget will respond to each of the possible events. The events that a selected widget recognizes are displayed on the Events tab of the Properties dialog. The event Values are initially undefined. Supply event routine names for only those events to which you want the application to respond.

### Opening the Properties dialog

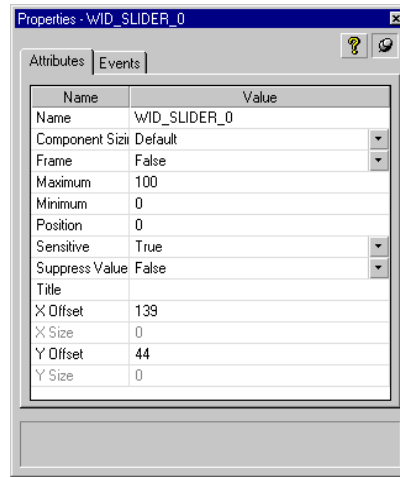
To open the Properties dialog for a widget:

Right-click on the widget in the IDL GUIBuilder window, and choose Properties from the menu.

*Or*

Select the widget, and choose Properties from the Edit menu.

These actions open a Properties dialog similar to the one shown in the following figure.



*Table 17-4: Properties Dialog for a Slider Widget*

The status area at the bottom of the Properties dialog, contains a description of the currently selected attribute or event. In addition, for each property that maps directly to an IDL keyword, there is a tool-tip that provides the name of the IDL keyword.

To display a tool-tip:

Place the cursor over the property name. The tool-tips are displayed only for properties that map to IDL keywords.

---

### Note

If you have multiple widgets selected in the IDL GUIBuilder window, the Properties dialog displays the properties for the primary selection, which is indicated by the darker, filled-in sizing handles around the widget. When you select multiple widgets, only one is marked as the primary selection.

---

To keep the Properties dialog on top:

Click the push pin button.

The Properties dialog will close as soon as it loses focus, unless you click the push pin button. If you click the push pin button, the Properties dialog stays on top and updates to reflect the properties of the currently selected widget.

To close the Properties dialog when the push pin is being used:

Click the push pin again, and the dialog will close when it loses focus.

*Or*

Press Escape while the dialog has focus.

*Or*

Click the close X in the upper right corner of the dialog.

Any changes you make to values in the Properties dialog are automatic; you will see the results of all visual changes immediately. For example, any changes you make to the alignment or column setting will change the layout position of the widget immediately.

All widgets share a common set of properties, and each widget has its own specific properties. These properties are arranged in the following order in the Properties dialog Attributes tab:

- The Name property
- An alphabetical list of common and widget-specific properties, combined

On the Properties dialog Events tab, the properties are displayed in alphabetical order with common and widget-specific events combined.

For information on the properties you can set for each widget, see [“Widget Properties”](#) on page 489.

## Entering Multiple Strings for a Property

There are several widget properties that you can set to multiple string values. The attribute’s Value field contains a popup edit control in which you can enter multiple strings.

To enter more than one string in the edit control, do one of the following:

Type in a string, then press Control+Enter, at the end of each line.

*Or*

Type in a string, then press Control+j, at the end of each line.

These actions move you to the next line. When you have entered the necessary string, press Enter to set the values.

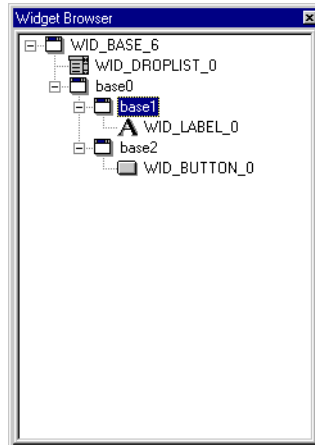
## Using the Widget Browser

The Widget Browser of the IDL GUIBuilder is a dialog window that presents the current GUI in a tree control. This presents the user with a different view into the GUI they are designing.

To start the Widget Browser:

Right-click on any component in a IDL GUIBuilder window, then choose Browse from the menu.

This action opens the Widget Browser, like the one shown in the following figure.



*Figure 17-5: Widget Browser*

The Widget Browser is helpful when you want to see your widget hierarchy and when you need to operate on overlapping widgets in your interface layout, which can happen when you design an interface to show or hide widgets on specific events. For an example that uses the Widget Browser for this purpose, see [“Controlling Widget Display”](#) on page 483.

---

**Note**

In the Widget Browser, there is no indication of defined menus.

---

You can expand the widget tree by clicking on the plus sign, or contract it by clicking on the minus sign.

When you select a widget in the hierarchy by clicking on it, the widget is selected in the IDL GUIBuilder window, and the Properties dialog updates to display the selected widget's properties.

Right-click on a component to display a context menu from which you can cut, copy, paste, or delete the widget. From the context menu, you can also open the Properties dialog and the Menu Editor, when appropriate. To delete a widget from the Widget Browser, use the context menu, or select a widget and press the Delete key.

To change a widget's Name property in the Widget Browser:

Select the widget name with two single clicks on the name. This action changes the name into an editable text box in which you can enter the new name. The **Name** property must be unique to the widget hierarchy.

For more information on other ways to operate on widgets, see [“Widget Operations”](#) on page 468.

## Using the Menu Editor

You can add menus to top-level bases or to buttons that have the Type property set to Menu. To define menus for your interface, use the Menu Editor, which is shown in the following figure with defined menus. This dialog allows you to define menus, menu items, submenu titles, and submenus, and all their associated event procedures.

For the instructions on how to define the menus shown in the following figure, see “Defining Menus for the Top-level Base” on page 442.

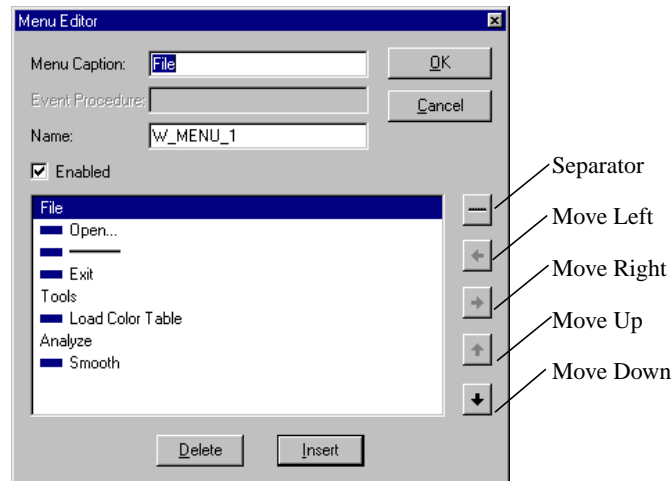


Figure 17-6: Menu Editor Dialog

To define basic menus, menu items, submenu titles, and submenus and their associated event procedures, to top-level bases, follow these general steps:

1. Open the Menu Editor by doing one of the following:  
 From the Edit menu, choose Menu.  
*Or*  
 Right-click on a top-level base, then choose Edit Menu.
2. To define a top-level menu in the Menu Editor, enter a Menu Caption, and click Insert. When you are defining menus for a top-level base, the top-level menus are aligned along the left edge of the menu list, and following the indentation indicates the nesting in the menu.

#### Note

The Menu Caption is the name that appears on the menubar. If you are defining a top-level menu for a base, you do not need to supply an Event Procedure. On button menus however, the button’s Label property acts as the top-level menu, and the first

level of menus in the editor serve as menu items, and thus need defined Event Procedures.

---

3. To define a menu item on a new line in the editor, click the right arrow, enter a Menu Caption and its associated Event Procedure, and then click Insert.

The Menu Caption is the name you want to appear on the menu. The Event Procedure is the name of the routine that will be called when the menu item is selected.

---

**Note**

For top-level bases, you must indent a line to make it a menu item and enable the Event Procedure field.

---

4. To define a submenu title, enter the Menu Caption, and click Insert. It is not necessary to define Event Procedures for submenu titles.
5. To define submenus to a submenu title, enter the Menu Caption and the Event Procedure, indent the item another level by using the right arrow, and click Insert. Enter the submenus you want at this level of indentation.
6. To define another top-level menu or menu item, enter the information, click the left arrow until the indentation is appropriate, and click Insert.
7. To define a separator, select a blank line, or select the line you want the separator after, then click the separator button (which has a line on it and is above the arrow buttons).
8. To save your defined menus, Click OK in the Menu Editor. When you do so, the menu items will appear on the top-level base. To test the display of the menus, click on them.

---

**Note**

Under Microsoft Windows, including the ampersand character (&) in the Menu Caption causes the window manager to underline the character following the ampersand, which is the keyboard accelerator. This functionality is supported in the Menu Editor. If you are designing an application to run on other platforms however, avoid the use of the ampersand in the Menu Caption.

---

To move a menu item to a new position:

Select the menu item, and click the up or down arrow on the right side of the dialog until the menu item is in the desired position. Then, click OK.

To add a menu item in the middle of existing menu items:

Select the line you want the new item to follow, then click Insert. This action adds a new line, for which you can enter a Menu Caption and Event Procedure.

To make a menu item display disabled initially:

Click the Enabled checkbox (to uncheck it). All menu items are enabled by default.

To delete a menu item:

Select the item, then click Delete.

To delete a menu:

Delete each contained menu item, then delete the top-level menu.

## Adding Menus to Buttons

You can also create buttons that contain menus. To add a menu to a button, follow these basic steps:

1. Click on the Button widget tool on the toolbar, then click on the top-level base area. This action creates a button of the default size.
2. Right-click on the button and choose Properties, which opens the Properties dialog.
3. In the Properties dialog, click on the Type attribute Value arrow, then choose Menu from the droplist.
4. Right-click on the button, then choose Edit Menu, which opens the Menu Editor. You can define the menu items and submenus with the Menu Editor, using the general steps described above.

---

### Note

For buttons, the button [Label](#) property acts as the top-level menu, and the first level of menus in the Menu Editor serve as menu items, and therefore require defined Event Procedures (unlike top-level menu items defined to bases).

---



5. After you have defined all the necessary menus, click OK. When you do so, the menus are saved, and, in the IDL GUIBuilder, the button Label property is displayed as the top-level menu.

To view menus on buttons:

Immediately after creating the menu (after clicking OK in the Menu Editor), click on the button, and the menus will display.

*Or*

At any other time, right-click on the button, and then choose Show Menu. After you do this, you can click on the button to view the menu items. To view the menus at any other time, choose Show Menu again.

## Using the Bitmap Editor

Use the Bitmap Editor to create 16 color bitmaps to be displayed on push buttons. The Bitmap Editor can read and write bitmap files (\*.bmp). Using the editor, you can create your own bitmaps, or you can open existing bitmap files and modify them.

IDL supplies a set of bitmap files you can use in the buttons of your applications. The files are always available for loading. The bitmaps are located in the following directory:

```
IDL_DIR\resource\bitmaps
```

## Placing a Color Bitmap on a Button

To display a bitmap on a button, follow these steps:

1. Right-click on the button widget, and choose Properties from the menu, which opens the Properties dialog for this button.
2. In the Type Value field, select Bitmap from the droplist.
3. In the Properties dialog, click on the arrow to the right of the Bitmap attribute, and do one of the following:

To place an existing bitmap in the button, choose Select Bitmap, and select a bitmap file from the Open dialog. This action displays the bitmap on the button. Note that when Bitmap type is selected, the label property changes to Bitmap.

*Or*

To edit an existing bitmap and place it in the button, choose Edit Bitmap, then select the bitmap file from the Open dialog. This opens the bitmap in the Bitmap Editor, and

assigns this as the bitmap to display on the button. It is displayed on the button when you save the file.

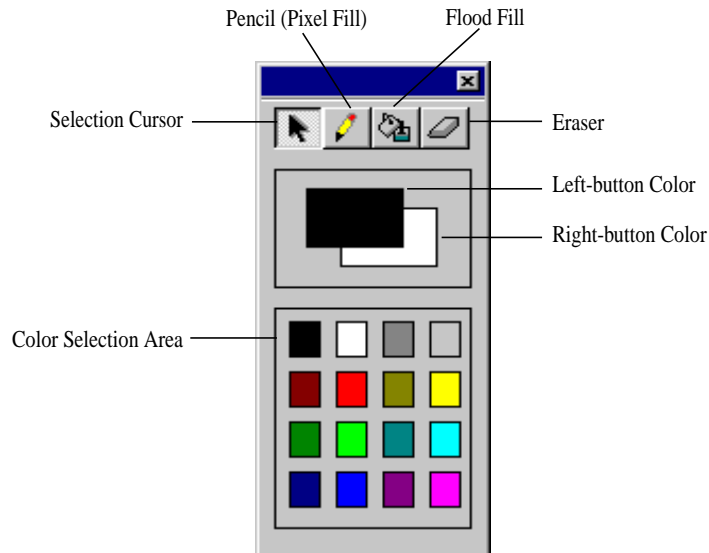
*Or*

To create a new bitmap and place it in a button, choose New Bitmap. This action opens the Bitmap Editor, which you can use to create the new bitmap. When you save the \*.bmp file, it is placed on the button.

When you complete one of these processes, the filename of the selected bitmap appears in the [Bitmap](#) field of the Properties dialog, and the bitmap is displayed on the button.

### Using the Bitmap Editor Tools

The Bitmap Editor tools allow you to select from the color palette, and then use the Pencil (pixel fill), the Flood fill (fill clear area), or the Eraser (clear or color areas). The Bitmap Editor tools are shown in the following figure.



*Figure 17-7: Bitmap Editor Tools*

You can select a color by clicking on it in the color selection tool, or you can select your primary colors, the left-button and right-button colors, and then click on a tool and draw on the bitmap canvas. You can change the primary color selections at any time.

To select the left mouse button color:

Left-click on the color in the color selection area.

To select a right mouse button color:

Right-click on the color in the color selection area.

To use the left color:

With a tool selected, click or press and drag the right mouse button on the bitmap canvas.

To use the right color:

With a tool selected, click or press and drag the left mouse button on the bitmap canvas.

To change the size of the bitmap:

Drag the bitmap canvas to the desired size.

# Widget Operations

The IDL GUIBuilder allows you to operate on widgets in many ways. You can select, deselect, move, cut, copy, paste, and delete widgets, and you can undo and redo operations. This section describes the following:

- [Selecting Widgets](#)
- [Moving and Resizing Widgets](#)
- [Cutting, Copying, and Pasting Widgets](#)
- [Deleting Widgets](#)
- [Undoing and Redoing Operations](#)

## Selecting Widgets

You can select a widget, then move it or resize it.

To select a widget:

Click on the widget.

To select more than one widget:

Press Shift and click on each widget.

*Or*

Press Control and click on the widgets. When you press Control, you can change the selection state by clicking again on the widget; pressing Control during selection allows you to toggle the selection state of a widget without affecting the selection state of any other widget.

*Or*

Press the left mouse button and drag out an area in the top-level base that includes the widgets you want to select. When you release the mouse button, widgets in the selection box are selected.

When you select multiple widgets, there is always one primary selection. The primary widget selection is indicated with the dark, filled-in selection handles. If you open the Properties dialog with multiple widgets selected, the properties displayed are those for the primary selection.

**Note**

When selecting multiple widgets, you can select only widgets that share the same base widget as their parent.

## Moving and Resizing Widgets

You can move widgets around in their parent base by dragging the widget to a new location or by using the arrow keys.

To move a widget to a new base; to give a widget a new parent base within the same top-level base:

Press Alt and drag and drop the widget on the new parent base.

*Or*

Right-click on the widget, choose Cut from the menu, right-click on the new base widget, and choose Paste from the menu.

**Note**

When you drag a widget to a new location, either in the same base or in another base, and the [Layout](#) attribute for the parent base is set to Column or Row, a blue line displays to indicate where the widget will be placed relative to other widgets in the base.

To resize a widget:

Click on a sizing handle, and drag to the desired size.

## Cutting, Copying, and Pasting Widgets

You can cut, copy, and paste widgets within the same base or to another base in another IDL GUIBuilder window, using the Edit menu items, or the toolbar buttons, or a context menu (opened with a right-click on the widget).

To cut or copy a selected widget, or to paste a widget from the clipboard:

Choose the appropriate operation from the Edit menu, or from the IDLDE toolbar.

*Or*

Right-click on the widget and select the appropriate operation from the menu. If you are pasting, right-click on the base widget you want to paste into.

*Or*

Select the widget and use standard windows keyboard shortcuts to cut, copy, or paste the widget.

---

**Note**

All cut or copied items are placed on a local clipboard, not on the system clipboard.

---

## Deleting Widgets

To delete a widget:

Select the widget and choose Delete from the Edit menu.

*Or*

Select the widget and press the Delete key.

*Or*

Right click on a widget and choose Delete from the menu.

## Undoing and Redoing Operations

In the IDL GUIBuilder, you can undo or redo unlimited operations between save procedures. If you save the resource file, the operations are cleared from memory.

To undo an operation:

Choose Undo from the Edit menu.

*Or*

Click the Undo button on the IDLDE toolbar.

*Or*

Press Control+z.

To redo an operation:

Choose Redo from the Edit menu.

*Or*

Click the Redo button on the IDLDE toolbar.

*Or*

Press Control+y.

# Generating Files

The IDL GUIBuilder generates the following two types of files:

- \*.prc files that contain the resource definitions for the interface definition as displayed in the IDL GUIBuilder.
- \*.pro files that contain the generated IDL source code. The generated \*.pro files are portable across all IDL-supported platforms.

## Generating Resource Files

The \*.prc files contain the resource definitions for the graphical interface. You can open \*.prc files in the IDL GUIBuilder and modify the interface at anytime. Do not attempt to modify this file directly.

To save a \*.prc file for the first time:

Choose Save or Save As from the IDLDE File menu. This opens the Save As dialog, which allows you to select a location and indicate a file name for the \*.prc file.

## Generating IDL Code

The IDL GUIBuilder can generate these two kinds of \*.pro IDL source code files:

- Widget definition code (\*.pro files).
- Event-handling code (\*\_eventcb.pro files).

To save both the widget code and the event handler \*.pro files:

From the IDLDE File menu, choose Generate .pro. This action opens the Save As dialog, which you can use to select a location and indicate a name for the widget code. The event code file name is based on the name specified for the widget code. For example, if you enter “app1.pro” in the File name field, the event code file will be named “app1\_eventcb.pro”.

### Note

---

Never modify the generated \*.pro interface file. If you decide to modify the application interface, use the IDL GUIBuilder, then regenerate the file. When you regenerate the widget code, the file is overwritten.

---

**Note**

---

When you save both files, IDL puts the RESOLVE\_ROUTINE procedure in the generated widget code. The procedure contains the name of the related \*\_eventcb.pro event-handler file so that it will be compiled and loaded with when you run the widget code.

---

**Notes on Generating Code a Second Time**

When you modify a interface and save the \*.prc file, it is overwritten, which should not be a problem. If you decide to change your interface, you will however need to regenerate the widget code and thus overwrite the \*.pro widget code file.

Note that if you regenerate either of the \*.pro files, they are overwritten. When writing code, you should modify *only* the generated event-handling file (\*\_eventcb.pro); and, you should never modify the generated widget code (the \*.pro file). This allows you to change the interface and regenerate the definition code without losing modifications in that file. This should simplify the procedures you need to take to update or change an interface.

Because it is modular, the event-handler code is simple to modify after you change the interface definitions. When you regenerate the IDL source code files, any new event handler code is appended to the end of the file.



# IDL GUIBuilder Examples

After you define your interface and generate IDL code using the IDL GUIBuilder, you will write the code that controls the application's behavior. You can modify the code, compile it, and run it using the IDLDE.

Generally, you will be writing the event-handler callbacks for the procedures located in the generated \*\_eventcb.pro file. While doing this, you might like to handle initialization states, have multiple GUIs work together, add compound widgets, or control widget display. For examples of how to handle these different types of events, see the following sections:

- [Understanding IDL GUIBuilder Event Handling Code](#)
- [Writing Event Callback Routines](#)
- [Handling Initialization Arguments](#)
- [Integrating Multiple Interfaces](#)
- [Adding Compound Widgets](#)
- [Controlling Widget Display](#)

## Understanding IDL GUIBuilder Event Handling Code

When using the IDL GUIBuilder, you assign event procedures to specific events using the Properties dialog Events tab. The calling sequence for the events that you set are added to the generated \*\_eventcb.pro event callback code.

The argument that is passed into the specified event routine depends on the type of event being processed. Creation, realization, and destruction event routines are usually passed the ID of the involved widget, and all other callback routines are passed the appropriate IDL widget event structure.

It is a normal operation in applications to change the attributes of the interface when a certain events occur. One method used in handling events for IDL GUIBuilder generated applications is the UNAME keyword, or the [Name](#) property, given to all created widgets. (In a programmatically created IDL application, this action is handled using information stored in a widget component's user value.)

When you create a widget in the IDL GUIBuilder, IDL gives it a name unique to the widget hierarchy to which it belongs. You can rename the widget using the [Name](#) property.

In the generated code, this name is specified by the UNAME keyword. Because these names are unique, you can use the WIDGET\_INFO function with the FIND\_BY\_UNAME keyword in your event callback routines to get the IDs of widgets in the interface application.

---

**Note**

For information on properties, see “Using the Properties Dialog” on page 457, and see “Widget Properties” on page 489.

---

## Writing Event Callback Routines

This short example shows how basic event processing works in IDL GUIBuilder-generated code. The example demonstrates how to use the FIND\_BY\_UNAME keyword to obtain the IDs of other widgets in the interface.

To create this simple example application, follow these steps:

1. From the IDLDE File menu, choose New, then choose GUI. This action opens a new IDL GUIBuilder window.
2. In the IDL GUIBuilder window, right-click on the contained top-level base, and choose Properties from the menu. This action opens a Properties dialog.
3. In the open Properties dialog, click the push pin button to keep the dialog open and on top.
4. On the Properties dialog Attributes tab, set the following for the top-level base:
  - Set the **Component Sizing** property to Default.
  - Set the **Layout** property to Column.
5. On the IDL GUIBuilder toolbar, click the Label widget button (the letter A).
6. Click on the top-level base area, which adds to the interface a label widget of the default.
7. With the label widget selected, set the following attributes in the Properties dialog:
  - In the **Name** field, enter “clock”.
  - Set the **Alignment** attribute to Center.
  - Set the **Component Sizing** attribute to Default.
  - In the **Text** field, enter “No Time Currently Available”.

8. On the IDL GUIBuilder toolbar, click the Button widget (the rectangle icon).
9. Click on the top-level base area, which adds a button widget to the interface.
10. With the button selected, set the following attributes in the Properties dialog:
  - In the **Label** field, enter “Time”.
11. In the Properties dialog, click the Events tab and do the following:
  - In the **OnButtonPress** field, enter “OnPress”.

Your interface definition should look like the one shown in the following figure.

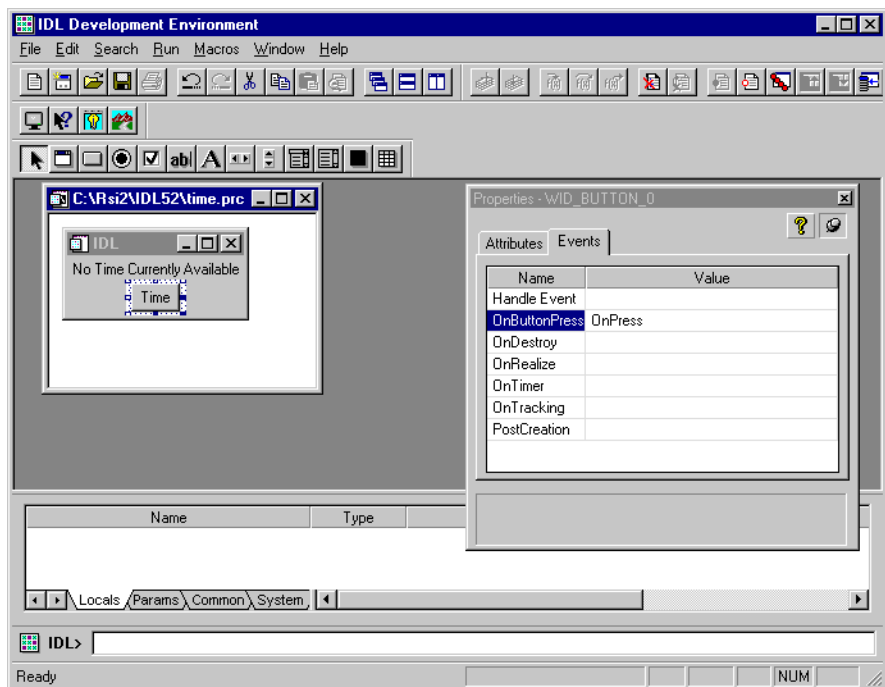


Figure 17-8: Handling Events Example Application

12. From the IDLDE File menu, choose Save, which opens the Save As dialog.
13. In the Save As dialog, select a location, enter “time.prc” in the File name field, and click Save. This action saves the interface definition to a resource file.

14. From the IDLDE File menu, choose Generate .pro which opens the Save As dialog.
15. In the Save As dialog, select the location, enter “time.pro” in the File name field, and click Save. This action saves the time.pro widget code file and the time\_eventcb.pro event callback code to the specified directory.
16. From the IDLDE File menu, choose Open, which launches the Open dialog.
17. In the Open dialog, locate and select the time\_eventcb.pro file, then click Open. This action opens the file in the IDLDE.
18. In the file, locate the OnPress event procedure place holder, and add the following IDL code to handle a button press, like this:

```

PRO OnPress, Event

    ; Get the widget ID of the label widget.
    Label = widget_info(Event.top, find_by_uname='clock')

    ; Set the value of the label widget to current time.
    widget_control, Label, set_value=Systemtime(0)

END

```

The first command gets the ID of the label widget by searching the widget hierarchy for a widget named “clock”. This is the name that you gave the label widget in the IDL GUIBuilder Properties dialog. Once the ID is found, the second command sets the value of the label widget to the current system time.

19. From the Run menu, choose, Compile time\_eventcb.pro, which saves and compiles the file.
20. To execute the program, enter the following at the IDL command prompt:

```
time
```

This compiles and runs the time.pro file. In the running application, you can press the Time button to cause the current time to be displayed in the label.

## Handling Initialization Arguments

You can provide runtime initialization information to the generated \*.pro widget code by modifying the \*\_eventcb.pro file. Keywords provided to the generated widget interface procedure are passed to the post creation routines using the \_EXTRA keyword.

If a routine is defined with the `_EXTRA` keyword parameter, you can add unrecognized keyword and value pairs, and the pairs are placed in an anonymous structure. The name of each unrecognized keyword becomes a tag name, and each value becomes the tag value.

You will use this feature most often when your application launches floating or modal dialogs, but the functionality is always available.

For example, if you want to display a dialog at the creation of an application, you would follow these basic steps:

1. Create an interface using the IDL GUIBuilder.
2. After creating the interface, open the Properties dialog for the top-level base and set the [PostCreation](#) event for the top-level base widget to a routine name, such as “OnCreate”.
3. Save the interface definition and generate the IDL source code
4. In the generated `*_eventcb.pro` event code file, locate the “OnCreate” routine place holder, which looks like this:

```
PRO OnCreate, wWidget, _EXTRA=_VWBExtra_

END
```

5. To process a specific keyword in this post creation routine, declare the keyword in the procedure statement and add the processing code to the procedure.

For example, to process the `DO_DIALOG` keyword in the defined `OnCreate` procedure, add the `DO_DIALOG` keyword to the procedure, and add the logic to handle it to the event callback routine. The completed procedure should look like this:

```
PRO OnCreate, wWidget, DO_DIALOG=DO_DIALOG, _EXTRA=_VWBExtra_

    ; If DO_DIALOG is set, display a simple message box.
    if( Keyword_Set(DO_DIALOG) )then $
        status = Dialog_Message("On Dialog Set")

END
```

6. Save the file, then compile and generate the application. To show the dialog at creation time, enter the following at the IDL command prompt:

```
<ProgramName>, /DO_DIALOG
```

## Integrating Multiple Interfaces

You can create multiple interfaces with the IDL GUIBuilder then integrate them to form the complete application hierarchy. This example shows you how to construct two interfaces and integrate them.

The first interface you will create is the main window, and it will consist of a simple push button that will launch a modal dialog. The second interface you will create is the modal dialog, and it will display a close button.

### Creating the Main Window

To create the main window, follow these steps:

1. From the IDLDE File menu, choose New, then choose GUI. This action opens a new IDL GUIBuilder window with a top-level base.
2. On the IDL GUIBuilder toolbar, click on the button widget button, then click on the top-level base. This action adds a button of the default size to the base. You can place the button anywhere in the base.
3. Right-click on the newly created button, and choose Properties from the context menu. This action opens a Properties dialog.
4. In the Properties dialog, click the push pin button to keep the dialog open and on top.
5. With the button selected, set the following in the Properties dialog:  
In the **Label** value field, enter “Modal Dialog”.
6. Click on the Properties dialog Events tab, and do the following:  
In the **OnButtonPress** value field, enter “OnPress”.
7. From the File menu, choose Save, which opens the Save As dialog.
8. In the Save As dialog, select a location, enter “maingui.prc” in the File name field, and click Save. This action saves the interface definition to an IDL resource file.
9. From the File menu, choose Generate .pro which opens the Save As dialog.
10. In the Save As dialog, select a location, enter “maingui.pro” in the File name field, and click Save. This action saves the maingui.pro widget code and the maingui\_evnetcb.pro event-handler code.
11. From the File menu, choose Open, which launches the Open dialog.

12. In the Open dialog, select the maingui\_eventcb.pro file, and click Enter.
13. In the open file, locate the OnPress event procedure place holder, then enter the code that launches the modal dialog, like this:

```
PRO OnPress, Event
    modalgui, group_leader=Event.top
END
```

You will create the “modalgui” dialog in the next set of steps. Note that you set the GROUP\_LEADER keyword here because the modal dialog requires it.

14. From the Run menu, choose Compile maingui\_eventcb.pro. This action saves and compiles the file.

## Creating the Modal Dialog

To create the modal dialog, follow these steps:

1. Open a new IDL GUIBuilder window.
2. In the IDL GUIBuilder window, select the top-level base, and set the following in the Properties dialog:
  - Set the **Modal** attribute to True.
  - In the **Title** field, enter “Modal Dialog”.
3. On the IDL GUIBuilder toolbar, click the button widget, then click on the top-level base. This action adds a button to the top-level base, and you can place it anywhere in the base.
4. With the new button selected, set the following in the Properties dialog:
  - In the **Label** field, enter “OK”.
5. Click on the Properties dialog Events tab, and do the following:
  - In the **OnButtonPress** value field, enter “OnModalPress”.
6. From the File menu, choose Save, which opens the Save As dialog
7. In the Save As dialog, select a location, enter “modalgui.prc” in the File name field, and click Save. This action saves the interface definition to an IDL resource file.
8. From the File menu, choose Generate .pro which opens the Save As dialog.

9. In the Save As dialog, select a location, enter “modalgui.pro” in the File name field, and click Save. This action saves the modalgui.pro widget code file and the modalgui\_eventcb.pro event callback file.
10. Open the modalgui\_eventcb.pro file and locate the OnModalPress procedure place holder. Then, add the following code so that the dialog closes when the button is pushed:

```
PRO OnModalPress, Event

    widget_control, Event.top, /destroy

END
```

11. Save and compile this file.

## Running the Example Application

Enter the following at the IDL command prompt:

```
maingui
```

This command runs the main window. You can press the Modal Dialog button, and the modal dialog is displayed. When you press the OK button on the modal dialog, the dialog exits.

## Adding Compound Widgets

The IDL GUIBuilder tools do not allow you to add a compound widget directly to your interface. You can, however, modify your event code to add a compound widget.

To add a compound widget to a IDL GUIBuilder generated interface, you will follow these basic steps:

1. Add the compound widget to the widget tree in a [PostCreation](#) event callback procedure.
2. Handle the events generated by the compound widget in the [Handle Event](#) callback function. Set this event function value for the base widget that will contain the compound widget.

## Adding a Compound Widget to an Interface

This example demonstrates how to add a compound widget to an application constructed with the IDL GUIBuilder. The application contains a label and a CW\_FSLIDER compound widget. In the running application, the values generated by CW\_FSLIDER will be displayed in the label widget.



To create this application, follow these steps:

3. From the IDLDE File menu, choose New, then choose GUI, which opens a new IDL GUIBuilder window with a top-level base.
4. Right-click on the base and choose Properties, which opens the Properties dialog for the top-level base.
5. In the Properties dialog, click the push pin button to keep the dialog on top.
6. In the Properties dialog of the top-level base, set the following properties:
  - Set the **Component Sizing** attribute to Default.
  - Set the **Layout** attribute to Column.
7. To add the label, click the label Widget button (the single letter) on the toolbar, then click on the top-level base. This action creates a label widget of the default size.
8. With the label selected, set the following in the Properties dialog:
  - In the **Name** value field, enter “label”.
  - Set the **Alignment** attribute to Center.
  - Set the **Component Sizing** attribute to Default.
  - In the **Text** value field, enter “000.000”.
9. Click the Base widget button on the toolbar (window icon), and click on the top-level base, This action adds a base to the top-level base.
10. With the new base widget selected, set the following in the Properties dialog:
  - Set the Component Sizing attribute to Default.
11. In the Properties dialog, click on the Events tab and set the following base widget event values:
  - In the **Handle Event** Value field, enter “HandleEvent”. This is the name of the function that will handle the compound widget events.
  - In the **PostCreation** Value field, enter “AddCW”. This is the name of the event routine that will create the compound widget.
12. To save the portable resource file, choose Save from the File menu, which opens the Save As dialog.

13. In the Save As dialog, select a location, enter “compound.prc” in the File name field, and click Save. This saves the interface definition to an IDL resource file.
14. From the File menu, choose Generate .pro which opens the Save As dialog.
15. In the Save As dialog, enter “compound.pro”, and click Save. This action generates the compound.pro widget code file and the compound\_eventcb.pro event-handler file.
16. From the IDLDE File menu, choose Open, which launches the Open dialog.
17. In the Open dialog, select the compound\_eventcb.pro event file, then click Open, which opens the file in the IDLDE.
18. In the file, locate the AddCW event routine place holder, and modify the code to add the CW\_FSLIDER compound widget to the base widget. The routine should look like this:

```
PRO AddCw, wWidget

    idslide = cw_fslider(wWidget, /suppress_value)

END
```

19. Add the event callback routines to the generated HandleEvent function. The function should look like this:

```
FUNCTION HandleEvent, Event

; Fslider event structure is an anonymous structure, so
; the following will return "" if it is from fslider.

if(Tag_Names(Event, /structure_name) eq "")then begin

    ; Get the id of the label widget using its name.
    id = widget_info(Event.top, find_by_uname='label')

    ; Set the value of the label, to the value in the slider.
    widget_control, id, set_value= $
        String(Event.value, format='(f5.2)')
    return, 0
    ; Halt event processing here.
end

return, Event
; By Default, return the event.

END
```

Note that the callback routine finds the label widget using the `FIND_BY_UNAME` keyword with the name value you gave the widget in the Properties dialog.

20. From the Run menu, choose `Compile compound_eventcb.pro`, which saves and compiles the file.

## Running the Example

To run the application, enter the following at the IDL command prompt:

```
compound
```

This action compiles and runs the application. In the running application, move the `CW_FSLIDER` and the value is placed in the label.

## Controlling Widget Display

This example demonstrates how to use the IDL GUIBuilder to create an interface that contains overlapping sub-bases containing different types of widgets. The example shows how you can display and hide overlapping controls in an interface created in the IDL GUIBuilder, and it incorporates using the Widget Browser. Note that this example is a slightly more complicated than the others.

This example constructs an interface with the following widgets:

- A droplist.
- A sub-base that contains two sub-bases:
  - One sub-base containing a text widget.
  - One sub-base containing a button.

The two contained sub-bases overlap and the visibility of each is controlled by the value selected in the droplist. When users select an item in the droplist, one sub-base is hidden and the other one is displayed.

## Creating the Interface

To create this application interface, follow these steps:

1. From the IDLDE File menu, choose `New`, then choose `GUI`. This action opens a new IDL GUIBuilder window with a top-level base.
2. Right-click on the top-level base, and choose `Properties` from the menu. This action opens a Properties dialog.

3. In the Properties dialog, click the push pin button to keep the dialog open and on top.
4. In the Properties dialog, do the following:
  - Set the **Component Sizing** attribute to Default.
  - Set the **Layout** attribute to Column.
5. On the IDL GUIBuilder toolbar, click on the droplist widget button, then click on the top-level base. This action creates a droplist in the base area.
6. With the droplist select, set the following in the Properties dialog:
  - In the **Title** value field, enter “Active Base”.
  - In the **Initial Value** field, click on the arrow. This displays an popup edit control in which you can enter “Base One”, press Control+Enter, enter “Base Two”, and press Enter.

---

**Note**

In the pop-up edit control for the **Initial Value** attribute, press Control+Enter to move to the next line. To set the values and close the popup edit control, press Enter.

---

7. Click on the Properties dialog Events tab, and do the following:
  - In the **OnSelectValue** field, enter “OnSelect”.
8. On the IDL GUIBuilder toolbar, click on the base widget button, then click on the top-level base. This action adds a base widget of the default size to the interface.
9. With the new base selected, set the following attributes in the Properties dialog:
  - In the **Name** value field, enter “base0”.
  - Set the **Frame** attribute to True.
10. On the IDL GUIBuilder toolbar, click on the base widget button, then click on the base you just added. This action adds a base widget to the “base0” widget.
11. With the newly-added base selected, set the following attributes in the Properties dialog:
  - In the **Name** value field, enter “base1”.

- Set the **Component Sizing** attribute to Explicit.
  - In the **X Offset** value field, enter “0”.
  - In the **X Size** value field, enter “200”.
  - In the **Y Offset** value field, enter “0”.
  - In the **Y Size** value field, enter “200”.
12. Right-click on a base, and choose Browse from the context menu. This action opens the Widget Browser.
  13. In the Widget Browser, right-click on base1, and choose Copy, which copies the widget to the local clipboard.
  14. In the Widget Browser, right-click on “base0”, and choose Paste, which pastes the copied base in to the “base0” widget. The new base is called “base1\_0”.
  15. In the Widget Browser, select “base1\_0”. This action selects the base in the IDL GUIBuilder window and updates the Properties dialog with the appropriate properties and values.
  16. With “base1\_0” selected, set the following attributes in the Properties dialog:
    - In the **Name** value field, enter “base2”.
    - Set the **Component Sizing** attribute to Explicit.
    - In the **X Offset** value field, enter “0”.
    - In the **X Size** value field, enter “200”.
    - In the **Y Offset** value field, enter “0”.
    - In the **Y Size** value field, enter “200”.
  17. From the File menu, choose Save, which opens the Save As dialog.
  18. In the Save As dialog, select a location, enter “visible.prc” in the File name field, and click Save. This action saves the interface definition.
  19. In the Widget Browser, select “base1”.
  20. With “base1” selected, set the following attribute in the Properties dialog:
    - Set the **Visible** attribute to False. This will hide “base1” and make “base2” visible.

21. On the IDL GUIBuilder toolbar, click the button widget button, then click on “base2” in the IDL GUIBuilder. This action adds a button to the base widget. Place the button anywhere in this base.
22. With the button selected, set the following attribute in the Properties dialog:
  - In the **Label** value field, enter “Button 2”.
23. In the Widget Browser, select “base2”, and set the following attribute in the Properties dialog:
  - Set the **Visible** attribute to False, which hides the base.
24. In the Widget Browser, select “base1”, and set the following attribute in the Properties dialog:
  - Set the **Visible** attribute to True, which shows the base.
25. On the IDL GUIBuilder toolbar, click the label widget button, then click on “base1”. This action adds a label to “base1”. You can place the label anywhere in this base.
26. With the label widget selected, set the following attribute in the Properties dialog:
  - In the **Text** value field, enter “Label 1”.
27. From the File menu, choose Save, which saves the changes to the visible.prc resource file.

The interface is now complete. It should look similar to the one shown in the following figure.

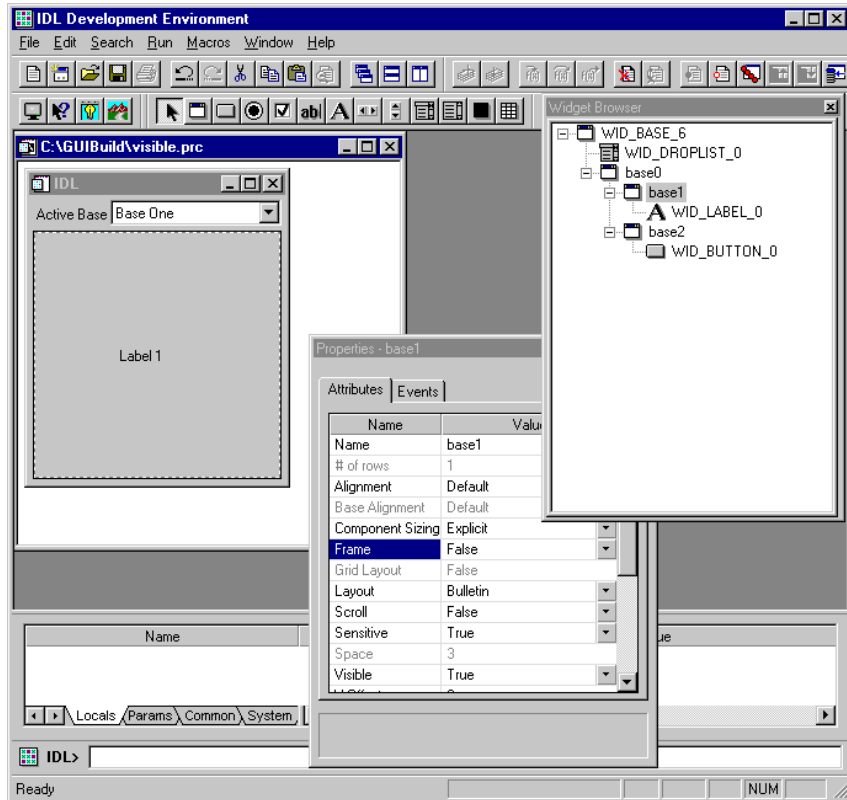


Figure 17-9: Visible Widgets Example Application

## Generating and Modifying the Code

To generate and modify the code, follow these steps:

1. From the File menu, choose Generate .pro which opens a Save As dialog.
2. In the Save As dialog, select a location, enter “visible.pro” in the File name field, and click Save. This action saves the visible.pro widget code file and the visible\_eventcb.pro event-handler file.
3. From the File menu, choose Open, which launches the Open dialog.
4. In the Open dialog, select the visible\_eventcb.pro file, and click Open. This action opens the file in the IDLDE.

5. In the `visible_eventcb.pro` file, locate the `OnSelect` event procedure place holder, then add the following code:

```
PRO OnSelect, Event

    ; Toggle the mapping of the two IDL sub-bases and
    ; get the Widget IDs of the two sub-bases.
    wBase1 = Widget_Info(Event.top, find_by_uname="base1")
    wBase2 = Widget_Info(Event.top, find_by_uname="base2")

    ; Now update the mapping.
    widget_control, wBase1, map=(Event.index eq 0)
    widget_control, wBase2, map=(Event.index eq 1)

END
```

The added IDL code gets the Widget IDs of the sub-bases that you created and sets the mapping (hide or show) of these bases depending on the selected value of the droplist.

6. From the Run menu, choose `Compile visible_eventcb.pro`, which saves and compiles the file.

## Running the Application

To run this application, enter the following at the IDL command prompt:

```
visible
```

This command executes the `visible` application. In the running application, you can change the selection in the droplist, and the action will change the displayed widget.



# Widget Properties

For each widget type, there is a set of attribute values and a set of event values you can set using the IDL GUIBuilder Properties dialog. When you select a widget in the IDL GUIBuilder window or in the Widget Browser, the Properties dialog is updated to contain the properties for the selected widget. These properties include those common to all widgets and those specific to the selected widget.

On the Attributes tab of the Properties dialog, the attributes are set to default values and are arranged in the following order:

- The [Name](#) property.
- An alphabetical list of common and widget-specific properties, combined.

On the Events tab, the possible events for a widget are listed in alphabetical order, with the common and the widget-specific events combined. By default, no event values are set initially. When you enter a routine name for an event property, you are responsible for making sure that event procedure exists. IDL does not validate the existence of the specified routine.

For information on how to open and use the Properties dialog, see [“Using the Properties Dialog”](#) on page 457.

The rest of this chapter describes the properties you can set for each widget:

- [Common Widget Properties](#)
- [Base Widget Properties](#)
- [Button Widget Properties](#)
- [Text Widget Properties](#)
- [Label Widget Properties](#)
- [Slider Widget Properties](#)
- [Droplist Widget Properties](#)
- [Listbox Widget Properties](#)
- [Draw Widget Properties](#)
- [Table Widget Properties](#)

# Common Widget Properties

There are several attribute and event property values you can set for all widgets. The attribute properties include the name of the widget and the sizing properties. The event properties include creation, realization, destruction, and tracking events.

The following sections describe the common properties:

- [Common Attributes](#)
- [Common Events](#)

## Common Attributes

These are the common attributes, which you can set for all widgets:

### Name

The Name attribute specifies the name of the component. This value can be any string that is unique to the widget hierarchy of the interface, but the string cannot contain spaces. For each widget you create in the IDL GUIBuilder, a default name is supplied, and this name is in the `WID_<TYPE>_<NUMBER>` format.

If you copy and paste a widget in the IDL GUIBuilder, the new widget is given a unique name based on the name of the one you copied. A number is added to the first widget's name, or an existing number is incremented.

You can use the Name value for the widget in your event callback routines. For example, you can use the specified name to find the widget, using the `FIND_BY_UNAME` keyword to the `WIDGET_INFO` function. Set the name for each widget to a name that makes sense to you; set the name value to something that is easy to remember and easy to use in your code.

In the generated \*.pro file, this value is specified with the `UNAME` keyword to the widget creation routines.

### Component Sizing

The Component Sizing keyword determines how the component is sized, which is by one of the following methods:

- **Default:** The widget is sized to a natural or implicit size. This is the default setting for the attribute. For example, a label widget's natural size is determined by the size of the text it is displaying with extra space for margins. The default size for each widgets is controlled by several things, including

displayed font size and the characteristics of the operating system displaying the interface.

- Explicit: The widget size is determined by several attributes, which include [Layout](#) for the base and its own [X Size](#), and [Y Size](#) attributes.

In the generated \*.pro widget file, this value is specified with the XSIZE, and YSIZE keywords to the widget creation routines.

---

**Note**

The default size of text widgets on Motif is based on the width of text, but the default size for text widgets on Windows and Macintosh is approximately 20 characters.

---

**Frame**

The Frame attribute determines if the widget will have a frame or border around it. These are the possible values:

- False: The widget will have no frame drawn around it. This is the default value.
- True: The widget will have a frame or border around it.

In the generated \*.pro widget file, this value is specified by the FRAME keyword to the widget creation routines.

---

**Note**

The Frame attribute is not available for top-level base widgets.

---

**Sensitive**

The Sensitive attribute determines if the selected widget is active or not active on startup. You can set this value to determine if the user can access and manipulate the widget immediately after creation. These are the possible values:

- True: The widget is initially displayed as enabled and accepts keyboard or mouse input and generates events. This is the default value.
- False: The widget is initially displayed as disabled and does not accept keyboard or mouse input. The appearance of most widgets change when the False value is set, but the appearance does not always change to indicate this state.

In the generated \*.pro file, this value is specified with the SENSITIVE keyword to the widget creation routines.

---

**Note**

To change the sensitivity of a widget after the widget is created, use the WIDGET\_CONTROL function with the SENSITIVE keyword.

---

**X Offset**

The X Offset attribute specifies the X offset of the component from its parent. The possible values for X Offset are  $o$  to  $n$ , in pixels; any number is valid. The [Y Offset](#) attribute specifies the Y offset.

In the generated \*.pro file, this value is specified with the XOFFSET keyword to the widget creation routines.

---

**Note**

The X Offset property value is *not* used with base widgets that have the [Layout](#) property set to Row or Column.

---

**X Size**

The X Size attribute specifies the width of the visible component in pixels. This property is disabled when [Component Sizing](#) is set to Default (and the default size is used). To enable this value, set Component Sizing to Explicit. The possible values for X Size are 0 to  $n$ , in pixels.

In the generated \*.pro file, this value is specified with the SCR\_XSIZE keyword to the widget creation routines.

---

**Note**

If you add scroll bars to a widget, use the widget-specific X Scroll property to set the width of the virtual area.

---

**Y Offset**

The Y Offset attribute specifies the Y offset of the component from its parent in pixels. The possible values for Y Offset are 0 to  $n$ , in pixels; any number is valid. The [X Offset](#) attribute specifies the X offset.

In the generated \*.pro file, this value is specified by the XOFFSET keyword to the widget creation routines.

---

**Note**

The Y Offset property value is *not* used with base widgets that have the [Layout](#) property set to Row or Column.

---

**Y Size**

The Y Size attribute specifies the height of the visible component in pixels. This property is disabled when [Component Sizing](#) is set to Default (and the default size is used). To enable this value, set Component Sizing to Explicit. The possible values for Y Size are 0 to *n*, in pixels.

In the generated \*.pro file, this value is specified with the SCR\_YSIZE keyword to the widget creation routines.

---

**Note**

If you add scroll bars to a widget, use the widget-specific Y Scroll property to set the height of the virtual area.

---

## Common Events

These are the common events, which you can set for all widgets (by default, no event values are initially set):

### Handle Event

The Handle Event value is the function name that is called when an event arrives from a widget that is rooted in a IDL GUIBuilder-created widget in the hierarchy. All events are sent to this event function, except for creation and destruction events.

For example, if you add a compound widget to an interface, using the [PostCreation](#) event procedure for a base widget, you should set the Handle Event value for that parent base widget (for the compound widget's parent widget). Then, you can handle all the events returned by the compound widget using this event function value.

In the generated \*\_eventcb.pro file, the event function place holder looks like this:

```
Function <Name>, Event  
  
    return, Event  
End
```

*Name* is the name of the event function you specify. *Event* is the returned event structure, which is specific to the widget event.

For an example of how to handle the generated Handle Event function, see [“Adding Compound Widgets”](#) on page 480.

## OnDestroy

The OnDestroy value is the routine name that is called when the widget is destroyed. In the generated \*\_eventcb.pro file, the event calling sequence looks like this:

```
PRO <RoutineName>, wWidget
```

*RoutineName* is the name of the event procedure you specify. *wWidget* is the IDL widget identifier.

## OnRealize

The OnRealize value is the routine name that is called automatically when the widget is realized. In the generated \*\_eventcb.pro file, the event calling sequence looks like this:

```
PRO <RoutineName>, wWidget
```

*RoutineName* is the name of the event procedure you specify. *wWidget* is the IDL widget identifier.

## OnTimer

The OnTimer value is the routine name that is called when a timer event is detected for a widget. In the generated \*\_eventcb.pro file, the event calling sequence looks like this:

```
PRO <RoutineName>, Event
```

*RoutineName* is the name of the event procedure you specify. *Event* is the returned event structure, which has the 3 standard event tags and looks like this:

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

You must set timer events for a widget, using the WIDGET\_CONTROL function. The code generated by the IDL GUIBuilder only routes the events.

## OnTracking

The OnTracking value is the routine name that is called when the widget receives a tracking event, which occurs when the mouse pointer *enters* or *leaves* the region of the widget. In the generated \*\_eventcb.pro file, the event calling sequence looks like this:

```
PRO <RoutineName>, Event
```

*RoutineName* is the name of the event procedure you specify. *Event* is the returned structure, which is of the following type:

```
{ WIDGET_TRACKING, ID:0L, TOP:0L, HANDLER:0L, ENTER:0 }
```

ENTER is 1 if the tracking event is an entry event, and 0 if it is an exit event.

### **PostCreation**

The PostCreation value is the routine name that is called after the widget is created, but before it is realized. In the generated \*\_eventcb.pro file, the calling sequence looks like this:

```
PRO <RoutineName>, wWidget
```

*RoutineName* is the name of the event procedure you specify. *wWidget* is the IDL widget identifier.

# Base Widget Properties

A base widget holds other widgets, including other base widgets. You can create groupings of widgets by using a base widget, thus forming a widget hierarchy.

When you open the IDL GUIBuilder, a top-level base is created, and you build your interface in this base. Top-level bases are a special class of the base widgets that are created without parent widgets; they act as the top-level parent in the widget hierarchy.

In the IDL GUIBuilder, you can add a menubar to the top-level base by using the Menu Editor.

In addition, you can make top-level bases *float* above their group leaders, with the [Floating](#) property, or you can make them *modal* dialogs, with the [Modal](#) property. Modal dialogs interrupt program execution until the user closes them. When you make a top-level base floating or modal, you must provide a group leader when calling the generated code, by using the `GROUP_LEADER` keyword.

When programming in IDL, you create base widgets using the `WIDGET_BASE` function. For more information, see [WIDGET\\_BASE](#) in the *IDL Reference Guide*.

For more information on the Menu Editor, see [“Using the Menu Editor”](#) on page 461.

---

## Note

A base widget’s layout is controlled by where you place it and the properties of its parent base.

---

## Base Widget Attributes

For base widgets, you can set common attributes and base-specific attributes. For a list attributes common to all widgets, see [“Common Attributes”](#) on page 490.

Some of the base widget attributes apply to top-level bases only, and this limitation is noted in the following list of base widget attributes:

### # of Rows/Columns

The # of Rows/Columns attribute specifies the number of Columns or Rows to use when laying out the base. This property is valid only when the [Layout](#) property is set to Column or Row. The possible values for this setting are 1 to  $n$ , and the default value is 1.



In the generated \*.pro file, this value is specified with the COLUMN or the ROW keyword to the widget creation routine.

For information on other properties that control the layout of contained widgets, see [Alignment](#), [Layout](#), [Space](#), [X Pad](#), and [Y Pad](#).

## Alignment

The Alignment attribute defines how components are aligned in the base. The way in which the value of this property affects the display of widgets depends on the value of the Layout property. The following is a list possible values for the Alignment property, and each value description includes information on how it works with the [Layout](#) settings:

- **Center:** Aligns the contained widgets with the center this parent base. This is the default value. For this setting to take effect, the Layout setting must be Row or Column. With Row set, the contained widgets are vertically centered. With Column set, the contained widgets are horizontally centered.
- **Top:** Aligns contained widgets with the top of this parent base. For this setting to take effect, the Layout setting must be Row.
- **Bottom:** Aligns the contained widgets with the bottom of this parent base. For this setting to take effect, the Layout setting must be Row.
- **Left:** Aligns the contained widgets with the left side of this parent base. For this setting to take effect, the Layout setting must be Column.
- **Right:** Aligns the contained widgets with the right side of this parent base. For this setting to take effect, the Layout setting must be Column.
- **Default:** Uses the default layout.

In the generated \*.pro file, these settings are specified with the BASE\_ALIGN\_CENTER, BASE\_ALIGN\_TOP, BASE\_ALIGN\_BOTTOM, BASE\_ALIGN\_LEFT, and BASE\_ALIGN\_RIGHT keywords to the widget creation routine.

For information on other properties that control the layout of contained widgets, see [# of Rows/Columns](#), [Layout](#), [Space](#), [X Pad](#), and [Y Pad](#).

## Allow Closing

The Allow Closing attribute determines if the top-level base can be closed by the user. By default, this value is set to True and the base can be closed. To make it so the top-level base cannot be close, set this value to False.

In the generated \*.pro file, this value is specified with the `TLB_FRAME_ATTR` keyword to the widget creation routine.

For information on other properties that control aspects of top-level bases, see the [Allow Moving](#), [Minimize/Maximize](#), [System Menu](#), and [Title Bar](#) properties.

---

**Note**

This property setting is used with top-level bases only. Note that this setting is only a hint to the window system and might be ignored by some window managers.

---

## Allow Moving

The Allow Moving attribute determines if the base can be moved. By default, this value is set to True, and the base can be moved. To suppress this behavior, set this value to False.

In the generated \*.pro file, this value is specified with the `TLB_FRAME_ATTR` keyword to the widget creation routine.

For information on other property settings that control aspects of top-level bases, see the [Allow Closing](#), [Minimize/Maximize](#), [System Menu](#), and [Title Bar](#) properties.

---

**Note**

This property setting is used with top-level bases only. Note that this setting is only a hint to the window system and might be ignored by some window managers.

---

## Floating

The Floating attribute determines if the top-level base is a floating base (always on top). By default, this setting is False, indicating that the base is *not* a floating base. To create a floating base, set this property to True.

If you make a top-level base floating, you must set the `GROUP_LEADER` keyword to a valid widget ID when calling the generated procedure.

In the generated \*.pro file, this value is specified with the `FLOATING` keyword to the widget creation routine.

---

**Note**

This property setting is used with top-level bases only.

---

## Grid Layout

The Grid Layout attribute determines if the base will have a grid layout, in which all columns have the same width, or in which all rows have the same height. These are the possible values:

- False: Columns or rows will not be the same size. This is the default value.
- True: Column widths or row heights are taken from the largest child widget. If you set this property to True, you must also set the [Layout](#) property to Column or Row and the [# of Rows/Columns](#) property to more than 1.

In the generated \*.pro file, this value is specified with the GRID\_LAYOUT keyword to the widget creation routine.

## Layout

The Layout attribute specifies how components are laid out in the base. These are the possible values:

- Bulletin: Indicates that you can position the widgets anywhere on the base. This is the default setting.
- Column: Indicated that widgets should be in columns. If you set this value, you should also set the [# of Rows/Columns](#) property and the [Alignment](#) property.
- Row: Indicated that widgets should be in rows. If you set this value, you should also set the [# of Rows/Columns](#) property and the [Alignment](#) property.

## Note

---

When using code generated by the IDL GUIBuilder on other non-Windows platforms, more consistent results are obtained by using a row or column layout for your bases instead of a bulletin board layout. By using a row or column layout, differences in the default spacing and decorations (e.g., beveling) of widgets on each platform can be avoided

---

The number of child widgets placed in each column or row is calculated by dividing the number of created child widgets by the number of columns or rows specified ([# of Rows/Columns](#)). When one column or row is filled, a new one is started.

The width of each column or the height of the row is determined by the largest widget in that column or row. If you set the [Grid Layout](#) property to True, all columns or rows are the same size; they are the size of the largest widget.

If you set the [Alignment](#) property for the base, the contained widgets are their “natural” size. If you do not set the Alignment property for the base or the child

widgets, all contained widgets will be sized to the width of the column or the height of the row.

For information on other properties that control the layout of contained widgets, see [# of Rows/Columns](#), [Alignment](#), [Space](#), [X Pad](#), and [Y Pad](#).

In the generated \*.pro file, this value is specified with the COLUMN or the ROW keyword to the widget creation routine.

---

**Note**

When you create a radio button or checkbox, it is created in a base, and you can add more radio buttons or checkboxes to that base (the added widgets must all be of the same type). The base in which radio buttons and checkboxes are created has a column layout setting, and buttons you add will be lined up in a column format.

---

**Minimize/Maximize**

The Minimize/Maximize attribute determines if the top-level base can be resized, minimized, and maximized. By default, this value is set to True. To disable this behavior, set this property to False.

In the generated \*.pro file, this value is specified with the TLB\_FRAME\_ATTR keyword to the widget creation routine.

For information on other property settings that control aspects of top-level bases, see the [Allow Closing](#), [Allow Moving](#), [System Menu](#), and [Title Bar](#) properties.

---

**Note**

This property setting is used with top-level bases only. Note that this setting is only a hint to the window system and might be ignored by some window managers.

---

**Modal**

The Modal attribute determines if this top-level base is a modal dialog. By default, this value is set to False. To make the base a modal dialog, set this property to True.

If you set the Modal property to True, you cannot set the [Scroll](#) property, and you cannot define a menu for the top-level base. In addition, the [Sensitive](#) common property and the [Visible](#) base widget property are also disabled.

If you make a top-level base a modal dialog, you must set the GROUP\_LEADER keyword to a valid widget ID in the generated procedure.

In the generated \*.pro file, this value is specified with the MODAL keyword to the widget creation routine.

---

**Note**

This property setting is used with top-level bases only.

---

**Scroll**

The Scroll attribute determines if the base widget will be support scrolling. By default, this property is set to False, and the base will not support scrolling. To give the widget scroll bars and allow for viewing portions of the widget contents that are not currently in the viewport area, set the Scroll property to True. In the IDL GUIBuilder, scroll bars on bases are live so that you can work on the entire virtual area of your application.

If you set the [Modal](#) property to True, you cannot set the Scroll property.

In the generated \*.pro file, this value is specified with the SCROLL keyword to the widget creation routine.

To set the size of the scrollable region, use the [X Scroll](#) and [Y Scroll](#) properties.

---

**Note**

For the Macintosh, if you set [X Size](#) or [Y Size](#) to a value less than 48, the base created with the Scroll property will be a minimum of 48x48. If you have not specified values for the X Size or Y Size property, the base will be set to a minimum of 66x66. If the base is resized, it will jump to the minimum size of 128x64.

---

**Space**

The Space attribute specifies the number of pixels between the contained widgets (the children) in a column or row [Layout](#). By default, this value is set to 3 pixels and that is the space between the contained widgets. Valid values for this property are 0 to *n* pixels.

In the generated \*.pro file, this value is specified with the SPACE keyword to the widget creation routine.

To set the space from the edge of the base, use the [X Pad](#) and [Y Pad](#) properties. For information on other properties that control the layout of contained widgets, see [# of Rows/Columns](#), [Alignment](#), and [Layout](#).

**Note**

---

You cannot set this property on a base containing radio buttons or checkboxes.

---

**System Menu**

The System Menu attribute determines if the system menu is displayed or suppressed on a top-level base. By default, this value is set to True, indicating that the system menu will be used. To suppress the menu, set this property to False.

In the generated \*.pro file, this value is specified with the TLB\_FRAME\_ATTR keyword to the widget creation routine.

For information on other property settings that control aspects of top-level bases, see the [Allow Closing](#), [Allow Moving](#), [Minimize/Maximize](#), and [Title Bar](#) properties.

**Note**

---

This property setting is used with top-level bases only.

---

**Title**

The Title attribute specifies the title of a top-level base. By default, this value is set to IDL, but you can change it to any string.

In the generated \*.pro file, this value is specified with the TITLE keyword to the widget creation routine.

**Note**

---

This property setting is used with top-level bases only.

---

**Title Bar**

The Title Bar attribute determines if the title bar will be displayed. By default, this value is set to True, and the title bar is displayed. To suppress the display of the title bar, set this value to False.

For interfaces running on the Macintosh, you cannot suppress the title bar because only modal dialogs use a window without a title bar. Suppressing the title bar would be contrary to Macintosh Human Interface Guidelines and would create an immovable window.

In the generated \*.pro file, this value is specified with the TLB\_FRAME\_ATTR keyword to the widget creation routine.

For information on other property settings that control aspects of top-level bases, see the [Allow Closing](#), [Allow Moving](#), [Minimize/Maximize](#), and [System Menu](#) properties.

---

**Note**

This property setting is used with top-level bases only, and it is only a hint to the window system and might be ignored by some window managers.

---

**Visible**

The Visible attribute specifies whether to show or hide the base component and its descendants. Show, the default value, specifies to display the hierarchy when realized. The Hide value specifies that the hierarchy should *not* be displayed initially. This mapping operation applies only to base widgets.

In the generated \*.pro file, this value is specified with the MAP keyword to the widget creation routine.

---

**Note**

If you set the [Modal](#) property to True, you cannot set this value.

---

**X Pad**

The X Pad attribute specifies the horizontal space (in pixels) between child widgets and the edges of rows or columns. By default, this value is set to 3 pixels, indicating that there are 3 pixels between the edge of the base and the contained widgets. Valid values for this property are 0 to  $n$  pixels.

In the generated \*.pro file, this value is specified with the XPAD keyword to the widget creation routine.

To set the space between widgets, use the [Space](#) property. For information on other properties that control the layout of contained widgets, see [# of Rows/Columns](#), [Alignment](#), [Layout](#), and [Y Pad](#).

---

**Note**

You cannot set this property for a base that contains radio buttons or checkboxes. In the IDL GUIBuilder, a base is created when you add a radio button or checkbox to an interface, and you can add more radio buttons or checkboxes to that base. When you add the buttons, they are lined up in a column format.

---

## X Scroll

The X Scroll attribute specifies the width in pixels of the base area, which includes the exposed as well as the virtual area. There is no default value set, but you can set this value to any number of pixels from 0 to  $n$ . To add scroll bars to the base, use the [Scroll](#) property, and to set the height of the scrollable base area, use the [Y Scroll](#) property.

In the generated \*.pro file, this value is specified with the XSIZE keyword to the widget creation routine.

---

### Note

To set the width of the displayed widget, use the [X Size](#) common property.

---

## Y Pad

The Y Pad attribute specifies the vertical space (in pixels) between child components and the edge of the base in a row or column [Layout](#). By default, this value is set to 3 pixels, indicating that there are 3 pixels between the edge of the base and the contained widgets. Valid values for this property are 0 to  $n$  pixels.

In the generated \*.pro file, this value is specified with the YPAD keyword to the widget creation routine.

To set the space between widgets, use the [Space](#) property. For information on other properties that control the layout of contained widgets, see [# of Rows/Columns](#), [Alignment](#), [Layout](#), and [X Pad](#).

---

### Note

You cannot set this property on a base containing radio buttons or checkboxes. In the IDL GUIBuilder, a base is created when you add a radio button or checkbox to an interface, and you can add more radio buttons or checkboxes to that base.

---

## Y Scroll

The Y Scroll attribute specifies the height in pixels of the base area, which includes the exposed as well as the virtual area. There is no default value set, but you can set this value to any number of pixels from 0 to  $n$ .

To add scroll bars to the base, use the [Scroll](#) property, and to set the width of the base area, use the [X Scroll](#) property.

In the generated \*.pro file, this value is specified with the YSIZE keyword to the widget creation routine.



**Note**


---

To set the height of the displayed widget, use the [Y Size](#) common property.

---

## Base Widget Events

For base widgets, you can set common event properties and base-specific event properties. By default, event values are *not* set. For a list of events common to all widgets, see “[Common Events](#)” on page 493.

The following is a list of event properties specific to base widgets:

### OnFocus

The OnFocus value is the routine name that is called when the keyboard focus of the base changes. In the generated \*\_eventcb.pro file, the event calling sequence looks like this:

```
PRO <RoutineName>, Event
```

*RoutineName* is the name of the event procedure you specify. *Event* is the returned event structure, which is returned when the keyboard focus changes and is of the following type:

```
{ WIDGET_KBRD_FOCUS, ID:0L, TOP:0L, HANDLER:0L, ENTER:0 }
```

ENTER returns 1 if the base is gaining the keyboard focus, and returns 0 if the base is losing the keyboard focus.

### OnKillRequest

The OnKillRequest value is the routine that is called when the user attempts to kill the top-level base widget. In the generated \*\_eventcb.pro file, the event calling sequence looks like this:

```
PRO <RoutineName>, Event
```

*RoutineName* is the name of the event procedure you specify. *Event* is the returned event structure, which is returned when a user tries to destroy the widget using the window manager and is of the following type:

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

Note that this event structure contains the standard three fields that all widgets contain.

**Note**

---

This event procedure is valid for top-level bases only.

---

**OnSizeChange**

The `OnSizeChange` value is the name the routine that is called when the top-level base has been resize. In the generated `*_eventcb.pro` file, the event calling sequence looks like this:

```
PRO <RoutineName>, Event
```

*RoutineName* is the name of the event procedure you specify. `Event` is the returned event structure, which is returned when the top-level base is resized by the user and is of the following type:

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

The `X` and `Y` fields return the new width of the base, not including any frame provided by the window manager.

**Note**

---

This event procedure is valid for top-level bases only.

---

# Button Widget Properties

In IDL, a button widget can be a button (push button), radio button, or checkbox.

A push button is activated by a single-click. Push buttons can be of any size. You can set the `Menu` property to `yes` for a button widget, and then it can contain a pull-down menu. When you do so, the `Label` is enclosed in a box to indicate that the button is a menu button.

Radio buttons have two states, set and unset, and they belong to a group that allows only one radio button selection for that group. The group is defined as all buttons contained in the same exclusive base widget. When a radio button in a base (in a group) is selected, any other button selection in that base is cleared. When you create a radio button in the IDL GUIBuilder, it is created in an exclusive base widget, and you can add only radio buttons to that base.

Checkboxes have two states, set and unset, and they are grouped in a non-exclusive base widget. This base allow for any number of checkboxes to be set at one time, and you can use single checkboxes in your interface. When you create a checkbox in the IDL GUIBuilder, it is created in an non-exclusive widget base, and you can add only checkboxes to this base.

When programming in IDL, you create push buttons, radio buttons, and checkboxes using the `WIDGET_BUTTON` function. For more information, see [WIDGET\\_BUTTON](#) in the *IDL Reference Guide*.

---

## Note

The bases in which radio buttons and checkboxes are created have the `Layout` attribute set to `column` so when you add more widgets they are lined up appropriately.

---

## Creating Multiple Radio Buttons or Checkboxes

To create several radio buttons or checkboxes in a base widget:

1. Click on the radio button or checkbox tool, and click on the location to add the button. This action creates a base with radio one button or checkbox in it.
2. Click on the radio button or checkbox tool, and click in the radio button or checkbox base area you just created. This action adds a radio button or checkbox to the base.

When you drop a button in a exclusive or non-exclusive base, the added buttons line up in columns; by default, these exclusive and non-exclusive bases have their [Layout](#) property set to Column.

3. Repeat step 2 until you have the desired number of buttons.
4. If you want to change the layout of the checkboxes or radio buttons, you can open the Properties dialog and set the [Layout](#) common property for the base widget to Row or Bulletin.
5. To set the properties for each button in the base, open the Properties dialog, click the push pin button to keep it on top, then click on each radio button or checkbox to set their individual properties.

## Button, Radio Button, and Checkbox Widget Attributes

For button widgets, you can set common attributes and button-specific attributes. For a list of common attributes, see “[Common Attributes](#)” on page 490. The following is a list of button widget attributes, which apply to push buttons, radio buttons, and/or checkboxes:

### Alignment

The Alignment attribute specifies how the text label is aligned in the button widget. These are the possible alignment values:

- Center: The label text is centered. This is the default value.
- Left: The label text is left-justified.
- Right: The label text is right-justified.

In the generated \*.pro file, this value is specified by the ALIGN\_CENTER, the ALIGN\_LEFT, or the ALIGN\_RIGHT keyword to the widget creation routine.

### Bitmap

The Bitmap attribute allows you to select a bitmap to be displayed in the push button, and it allows you to access the Bitmap Editor to create or modify a bitmap file (\*.bmp file). This value applies only to buttons (not to radio buttons or checkboxes).

To set this value:

Set the [Type](#) value to Bitmap, then the Bitmap attribute displays in the Properties dialog. When the button type is “Bitmap”, you can set the Bitmap attribute to the path and name of the bmp file.

When you click on the arrow in the Bitmap attribute Value field, you can choose from the following options:

- **Select Bitmap:** Launches an Open dialog that you can use to locate and select the existing \*.bmp file to be placed in the button.
- **Edit Bitmap:** Launches an Open dialog that you can use to locate and select the existing \*.bmp file to be opened in the Bitmap Editor. You can modify the bitmap and save it. The bitmap is then displayed in the button.
- **New Bitmap:** Opens the Bitmap Editor which you can use to create and save a bitmap. When you save the new bitmap, it is displayed in the button.

In the generated \*.pro file, this value is specified with the VALUE and Bitmap keyword to the widget creation routine.

For information on using the Bitmap Editor, see [“Using the Bitmap Editor”](#) on page 465.

## Label

The Label attribute specifies the text label for a button. If you set the **Type** attribute to **Bitmap** (for push buttons only), this value is not displayed. For radio buttons and checkboxes, the label value is the text string displayed next to the button. By default, this value is set to Button, and you can change it to any string.

In the generated \*.pro file, this value is specified with the VALUE keyword to the widget creation routine.

## No Release

The No Release attribute enables and disables the dispatching of button release events for radio buttons and checkboxes. Normal buttons do not generate events when released, but radio buttons and checkboxes can return separate events for the select and release actions. These are the possible values:

- **True:** The release event is not returned; only the select event is returned. This is the default setting.
- **False:** Both the release and select events are returned.

In the generated \*.pro file, this values is specified with the NO\_RELEASE keyword to the widget creation routine.

## Note

---

The No Release property is for radio buttons and checkboxes only.

---

## Type

The Type attribute specifies if a push button is a plain push button, a menu button, or a bitmap button. This attribute applies only to push buttons (not to radio buttons or checkboxes). These are the possible values:

- Push: The button widget is a plain push button. This is the default value.
- Menu: The button contains a menu. After you select this value, you can right-click on the button widget, choose Edit Menu, and define a menu to display, using the Menu Editor.
- Bitmap: The button displays a bitmap, which you would use to create a toolbar for example. If you change the Type value to Bitmap, the [Bitmap](#) property is displayed and you can select, modify, or create a bitmap to display on the button.

In the generated \*.pro file, this value is specified with the MENU or VALUE keywords to the widget creation routine.

## Button, Radio Button, and Checkbox Widget Events

For button widgets, you can set common event properties and button-specific event properties. By default, event values are *not* set. For a list of events common to all widgets, see “[Common Events](#)” on page 493.

The following is the event property specific to button widgets; it applies to push buttons, radio buttons, and checkboxes:

### OnButtonPress

The OnButtonPress value is the routine that is called when the button is pressed, or when a button is released for a radio button or checkbox button. In the generated \*\_eventcb.pro file, the event calling sequence looks like this:

```
PRO <RoutineName>, Event
```

*RoutineName* is the name of the event procedure you specify. Event is the returned event structure, which is of the following type:

```
{ WIDGET_BUTTON, ID:0L, TOP:0L, HANDLER:0L, SELECT:0 }
```

SELECT is set to 1 if the button was set, and 0 if released. Push buttons do not generate events when released, so SELECT will always be 1 for a push button. However, radio buttons and checkboxes are toggle buttons, and thus return separate events for the set and the release actions. To control whether or not release events are returned, set the [No Release](#) property.

# Text Widget Properties

Use text widgets to display text, and optionally, use them to accept textual input from users. The text widgets can have one or more lines, and if necessary, the widget can contain scroll bars to allow for viewing longer text.

When programming in IDL, you create text widgets using the `WIDGET_TEXT` function. For more information, see [WIDGET\\_TEXT](#) in the *IDL Reference Guide*.

---

**Note**

Use text widgets for displaying large amounts of text, or when you want the user to be able to edit the text. Use label widgets to display single-line labels that the user cannot edit.

---

## Text Widget Attributes

For text widgets, you can set common attributes and text-specific attributes. For a list of common attributes, see “[Common Attributes](#)” on page 490. The following are the attributes specific to text widgets:

### Editable

The Editable attribute determines if the text widget is editable or not. By default, this value is set to `False`, which means the text widget is not editable. To make the text widget editable, set this value to `True`.

In the generated `*.pro` file, this value is specified with the `EDITABLE` keyword to the widget creation routine.

### Height

The Height attribute specifies the height of the text widget in text lines. Valid values for this attribute are 1 to  $n$ . The default value, is 1, or one text line.

Note that the physical height of the text widget depends on the value of the Height attribute and on the size of the font used. The default font size is used, unless you modify your generated code to use a different font, and the default font size is platform specific.

In the generated `*.pro` file, this value is specified by the `YSIZE` keyword to the widget creation routine.

## Initial Value

The Initial Value attribute specifies the initial array of values that are placed in the text widget. You can enter either a string or an array of strings.

To enter more than one string in the Value field:

Type in a string, then press Control+Enter (at the end of each line). This action moves you to the next line. When you have entered the strings you want, press Enter to set the values.

In the generated \*.pro file, this value is specified by the VALUE keyword to the widget creation routine.

---

### Note

Variables returned by the GET\_VALUE keyword to WIDGET\_CONTROL are always string arrays, even if a scalar string is specified in the call to WIDGET\_TEXT.

---

## Scroll

The Scroll attribute determines if the text widget displays scroll bars. By default, this value is set to False, which indicates that no scroll bars will be displayed. To have the text widget display scroll bars, set this value to True.

In the generated \*.pro file, this value is specified by the SCROLL keyword to the widget creation routine.

## Width

The Width attribute specifies the width of the text widget in characters. Valid values for this attribute are 0 to  $n$ . By default, Width is set to 0, which indicates that default IDL sizing should be used when, as long as default [Component Sizing](#) is also set.

Note that the physical width of the text widget depends on the value of the Width attribute and on the size of the font used. The default font size varies according to your windowing system. On Windows and Macintosh, the default size is roughly 20 characters. On Motif, the default size depends on the system default.

In the generated \*.pro code, this value is specified with the XSIZE keyword.

## Word Wrapping

The Word Wrapping attribute determines whether a scrolling or multi-line text widgets should automatically break lines between words to keep the text from extending past the right edge of the text display area. By default this value is set to



False, and carriage returns are not automatically entered; the value of the text widget will remain a single-element array unless. To have the text widget enter carriage returns at the end of lines, change this value to True.

In the generated \*.pro code, this value is specified with the WRAP keyword.

## Text Widget Events

For text widgets, you can set common event properties and text-specific event properties. By default, event values are *not* set. For a list of events common to all widgets, see “[Common Events](#)” on page 493.

You can set the following event values for text widgets:

### OnDelete

The OnDelete value is the routine that is called when text is deleted from the text widget. To set this event value, you must set the [Editable](#) attribute to True.

In the generated \*\_eventcb.pro file, the calling sequence looks like this:

```
PRO <RoutineName>, Event
```

*RoutineName* is the name of the event procedure you specify. *Event* is the returned event structure, which is returned when any amount of text is deleted from a text widget. The event structure is of the following type:

```
{ WIDGET_TEXT_DEL, ID:0L, TOP:0L, HANDLER:0L, TYPE:2, OFFSET:0L,
  LENGTH:0L }
```

OFFSET is the (zero-based) character position of the first character to be deleted, and it is also the insertion position that will result when the characters have been deleted. LENGTH gives the number of characters deleted, where 0 (zero) indicates that no characters were deleted.

### OnFocus

The OnFocus value is the routine that is called when the keyboard focus changes. In the generated \*\_eventcb.pro event code, the calling sequence looks like this:

```
PRO <RoutineName>, Event
```

*RoutineName* is the name of the event procedure you specify. *Event* is the returned structure, which is of the following type:

```
{ WIDGET_KBRD_FOCUS, ID:0L, TOP:0L, HANDLER:0L, ENTER:0 }
```

ENTER returns 1 if the text widget is gaining the keyboard focus, or 0 if the text widget is losing the keyboard focus.

## OnInsertCh

The `OnInsertCh` value is the routine that is called when a single character is inserted in the widget. To set this event value, you must set the `Editable` attribute to `True`.

In the generated `*_eventcb.pro` file, the calling sequence looks like this:

```
PRO <RoutineName>, Event
```

*RoutineName* is the name of the event procedure you specify. `Event` is the returned event structure, which is returned a single character is typed or pasted into a text widget by a user. The event structure is of the following type:

```
{ WIDGET_TEXT_CH, ID:0L, TOP:0L, HANDLER:0L, TYPE:0, OFFSET:0L,
  CH:0B }
```

`OFFSET` is the (zero-based) insertion position that will result after the character is inserted. `CH` is the ASCII value of the character.

## OnInsertString

The `OnInsertString` value is the routine that is called when a text string is inserted in the text widget. To set this event value, you must set the `Editable` attribute to `True`.

In the generated `*_eventcb.pro` file, the calling sequence looks like this:

```
PRO <RoutineName>, Event
```

*RoutineName* is the name of the event procedure you specify. `Event` is returned structure, which is of the following type, which is returned when multiple characters are inserted in to text widget:

```
{ WIDGET_TEXT_STR, ID:0L, TOP:0L, HANDLER:0L, TYPE:1, OFFSET:0L,
  STR:'' }
```

`OFFSET` is the (zero-based) insertion position that will result after the text is inserted. `STR` is the string to be inserted.

## OnTextSelect

The `OnTextSelect` value is the routine that is called when text is selected in the text widget. To set this event value, you must also set the `Editable` attribute to `True`.

In the generated `*_eventcb.pro` file, the calling sequence looks like this:

```
PRO <RoutineName>, Event
```

*RoutineName* is the name of the event procedure you specify. `Event` is the returned event structure, which is returned when an area of text is selected. The event structure is of the following type:

```
{ WIDGET_TEXT_SEL, ID:0L, TOP:0L, HANDLER:0L, TYPE:3, OFFSET:0L,  
  LENGTH:0L }
```

This event announces a change in the insertion point. `OFFSET` is the (zero-based) character position of the first character selected, which can also be the insertion position. `LENGTH` gives the number of characters involved, where zero indicates that no characters are selected.

---

**Note**

Text insertion, text deletion, or any change in the current insertion point causes any current selection to be lost. In such cases, the loss of selection is implied by the text event reporting the insert, delete, or movement event, and a separate zero length selection event is *not* sent.

---

# Label Widget Properties

Label widgets display static text. They are similar to single-line text widgets, but they are optimized for small labeling purposes.

There are not label widget-specific event properties.

When programming in IDL, you create label using the `WIDGET_LABEL` function. For more information, see [WIDGET\\_LABEL](#) in the *IDL Reference Guide*.

---

**Note**

Use label widgets to display single-line labels that you do not want the user to be able to edit. Use text widgets for displaying larger amounts of text, or text that you want the user to be able to edit.

---

## Label Widget Attributes

For label widgets, you can set common attributes and label-specific attributes. For a list of common attributes, see “[Common Attributes](#)” on page 490. These are the label widget attributes:

### Alignment

The Alignment attribute specifies how label `Text` is aligned. These are the possible values:

- Left: The text is left-justified. This is the default value.
- Center: The text is centered.
- Right: The text is right-justified.

In the generated \*.pro file, this value is specified with the `ALIGN_CENTER`, the `ALIGN_RIGHT`, or the `ALIGN_LEFT` keyword to the widget creation routine.

### Text

The Text attribute specifies the text string that is displayed in the label widget. By default, this value is set to Label, and you can set it to any string.

In the generated \*.pro file, this value is specified with the `VALUE` keyword to the widget creation routine.

## Label Widget Events

There are *no* events specific to Label widgets. For a list of the common widget events, see [“Common Events”](#) on page 493.

# Slider Widget Properties

Horizontal or vertical slider widgets allow for the selection of a value within a range of possible integer values. A slider widget is a rectangular region representing a range of values, with a sliding pointer inside that indicates or selects the current value. This sliding pointer can be manipulated by the user dragging it with the mouse, or within IDL code.

When programming in IDL, you create horizontal or vertical slider widgets using the `WIDGET_SLIDER` function. See [WIDGET\\_SLIDER](#) in the *IDL Reference Guide*.

## Horizontal and Vertical Slider Widget Attributes

For slider widgets, you can set common attributes and slider-specific attributes. For a list of common attributes, see “[Common Attributes](#)” on page 490. The following is a list of slider attributes:

### Maximum Value

The Maximum Value attribute specifies the maximum range value for the slider. The default value is 100, but you can set this property to any integer. This value works with the [Minimum Value](#) property.

In the generated \*.pro file, this value is specified with the `MAXIMUM` keyword to the widget creation routine.

### Minimum Value

The Minimum Value attribute specifies the minimum range value of the slider. The default value is 0, but you can set this property to any integer. This property works with the [Maximum Value](#) property.

In the generated \*.pro file, this value is specified with the `MINIMUM` keyword to the widget creation routine.

### Position

The Position attribute specifies the initial value position of the slider. By default this is set to 0, so the initial position will be at 0. You can set this value to any integer within the range of the [Maximum Value](#) and [Minimum Value](#) attribute settings.

In the generated \*.pro file, this value is specified with the `VALUE` keyword to the widget creation routine.

## Suppress Value

The Suppress Value attribute controls the display of the current slider value. Sliders work only with integer units. You can use this property to suppress the actual value of a slider so that a program can present the user with a slider that seems to work in other units (such as floating-point) or with a non-linear scale. By default, this value is set to False, indicating that the current slider values, in integer units, should be displayed. To suppress the display of the current values, set this property value to True.

In the generated \*.pro file, this value is specified with the SUPPRESS\_VALUE keyword to the widget creation routine.

## Title

The Title attribute specifies the label or title that is associate with the slider widget. By default, this is not set; it is an empty string. You can set the title to any string.

In the generated \*.pro file, this value is specified with the TITLE keyword to the widget creation routine.

## Horizontal and Vertical Slider Widget Events

For slider widgets, you can set common event properties and slider-specific event properties. By default, event values are *not* set. For a list of events common to all widgets, see “[Common Events](#)” on page 493.

This is the event property specific to slider widgets:

## OnChangeValue

The OnChangeValue specifies the routine that is called when the value of the slider is changed. When you set this event value, the calling sequence looks like this in the generated \*\_eventcb.pro file:

```
PRO <RoutineName>, Event
```

*RoutineName* is the name of the event procedure you specify. *Event* is the returned event structure, which is returned when a slider is moved. The event structure is of the following type:

```
{ WIDGET_SLIDER, ID:0L, TOP:0L, HANDLER:0L, VALUE:0L, DRAG:0 }
```

VALUE returns the new value of the slider. DRAG returns integer 1 if the slider event was generated as part of a drag operation, or zero if the event was generated when the user had finished positioning the slider. Note that the slider widget only generates

events during the drag operation if the DRAG keyword is set, and if the application is running on Motif. That is, in most cases, DRAG will return zero.



# Droplist Widget Properties

Droplist widgets display a single entry from a list of possible choices. To choose from the list, click the droplist, then click on the item in the list. On Motif operating systems, the droplist widget looks like a button, which when clicked displays the drop-down list.

When programming in IDL, you create droplist widgets using the `WIDGET_DROPLIST` function. For more information, see [WIDGET\\_DROPLIST](#) in the *IDL Reference Guide*.

## Droplist Widget Attributes

For droplist widgets, you can set common attributes and droplist-specific attributes. For a list of common attributes, see “[Common Attributes](#)” on page 490. These are the droplist attributes:

### Initial Value

The Initial Value attribute specifies the initial list of values that are placed in the droplist widget. The initial value of a droplist can be a scalar string, or it can be a list of strings. By default, this value is not set, and the droplist is empty.

To enter more than one string in the Value field:

Type in a string, then press Control+Enter (at the end of each line). This action moves you to the next line. When you have entered as many strings as you want, press Enter to set the values.

In the generated \*.pro file, this value is specified with the `VALUE` keyword to the widget creation routine.

### Title

The Title attribute specifies the title string, or label, for the droplist. This value can be any string. By default, this value is set to `NULL`.

In the generated \*.pro file, this value is specified by the `TITLE` keyword to the widget creation routine.

## Droplist Widget Events

For droplist widgets, you can set common event properties and droplist-specific event properties. By default, event values are *not* set. For a list of events common to all widgets, see “[Common Events](#)” on page 493.

This is the event property specific to droplist widgets:

### **OnSelectValue**

The `OnSelectValue` specifies the routine that is called when a droplist item is selected. When a user selects an item from a droplist, the widget deselects the previously selected item, changes the visible item on the droplist, and generates an event.

When you set this event value, the calling sequence looks like this in the generated `*_eventcb.pro` file:

```
PRO <RoutineName>, Event
```

*RoutineName* is the name of the event procedure you specify. `Event` is the returned event structure, which and is returned when a user selects an item from a droplist and is of the following type:

```
{ WIDGET_DROPLIST, ID:0L, TOP:0L, HANDLER:0L, INDEX:0L }
```

`INDEX` returns the index of the selected item. This value can be used to index the array of names originally used to set the widget's value.

---

### **Note**

On some platforms, when a droplist widget contains only one item and the user selects the again, the action does not generate an event. Events are always generated on selection actions if the list contains multiple items.

---

# Listbox Widget Properties

The listbox displays a list of text items from which a user can select, by clicking on them. The listboxes have vertical scroll bars to allow viewing of a long list of items.

When programming in IDL, you create listbox widgets using the `WIDGET_LIST` function. For more information, see [WIDGET\\_LIST](#) in the *IDL Reference Guide*.

## Listbox Widget Attributes

For listbox widgets, you can set common attributes and listbox-specific attributes. For a list of common attributes, see “[Common Attributes](#)” on page 490. These are the listbox widget attributes:

### Height

The Height attribute specifies the height of the listbox based on the number of lines that are visible. The possible values for the attribute are 1 to  $n$ . By default, Height is set to 1, which indicates the default size of one line will be used.

Note that the final size of the widget may be adjusted to include space for scroll bars, which are not always visible, so the listbox might be slightly larger than specified.

In the generated \*.pro file, this value is specified with the `YSIZE` keyword to the widget creation routine.

### Initial Value

The Initial Value attribute specifies the initial list of values that are placed in the list widget. By default, the list is empty, but you can set this value to a scalar string or a list of strings. List widgets are sized based on the length (in characters) of the longest item specified in the array of values.

To enter more than one string in the Value field:

Type in a string, then press Control+Enter (at the end of each line). This action moves you to the next line. When you have entered as many strings as you want, press Enter to set the values.

In the generated \*.pro file, this value is specified by the `VALUE` keyword to the widget creation routine.

### Multiple

The Multiple attribute determines if the user can select multiple list items. By default, the setting is `False`, which allows for only one selection. To enable multiple list item

selection, set this value to True. Multiple selections are handled using the method appropriate to the platform the application is running on.

In the generated \*.pro file, this value is specified with the MULTIPLE keyword to the widget creation routine.

## Width

The Width attribute specifies the width of the listbox in characters. The possible values for the attribute are 0 to *n*. By default, Width is set to 0, which indicates that default sizing will be used, as long as the [Component Sizing](#) attribute is set to default.

By default, IDL sizes widgets to fit the situation. However, if the desired effect is not produced, use explicit Component Sizing with the Width property to set your own sizing. The final size of the widget may be adjusted to include space for the scroll bar, which is not always visible, so your widget may be slightly larger than specified.

In the generated \*.pro file, this value is specified with the XSIZE keyword to the widget creation routine.

## Listbox Widget Events

For listbox widgets, you can set common event properties and listbox-specific event properties. By default, event values are *not* set. For a list of events common to all widgets, see [“Common Events”](#) on page 493.

The following is the event property specific to listbox widgets:

### OnSelectValue

The OnSelectValue specifies a valid IDL routine name that is called when a list item is selected. When a user clicks on an item in the listbox to select the item, an event is generated.

When you set this event value, the calling sequence looks like this in the generated \*\_eventcb.pro file:

```
PRO <RoutineName>, Event
```

*RoutineName* is the name of the event procedure you specify. *Event* is the returned event structure, which is of the following type:

```
{ WIDGET_LIST, ID:0L, TOP:0L, HANDLER:0L, INDEX:0L, CLICKS:0L }
```

The first three fields are the standard fields found in every widget event. INDEX returns the index of the selected item. This index can be used to subscript the array of names originally used to set the widget's value. CLICKS returns either 1 or 2,

depending on how the list item was selected. If the list item is double-clicked, `CLICKS` is set to 2.

---

**Note**

If you are writing a widget application that requires the user to double-click on a list widget, you will need to handle two events. The `CLICKS` field will return a 1 on the first click and a 2 on the second click.

---

# Draw Widget Properties

Draw widgets are rectangular regions that IDL treats as standard graphics windows. Use draw widgets to display either IDL Direct graphics or IDL Object graphics, depending on the value of the [Graphics Type](#) property. You can direct any graphical output that can be produced by IDL to one of these widgets, either by using the [WSET](#) function or by using the object reference of a draw widget's `IDLgrWindow` object.

Draw widgets can contain scroll bars that allow for viewing of a graphical region larger than the area containing the widget.

When programming in IDL, you create draw area widgets using the `WIDGET_DRAW` function. For more information, see [WIDGET\\_CONTROL](#) in the *IDL Reference Guide*.

## Draw Area Widget Attributes

For a draw area widget, you can set common attributes and draw area-specific attributes. For a list of common attributes, see [“Common Attributes”](#) on page 490. These are the draw area-specific attributes:

### Color Model

The Color Model attribute specifies the color model that should be used for displaying information on the draw widget. This property value is used only when the [Graphics Type](#) property is set to Object, for IDL Object Graphics. These are the possible values for the Color Model attribute:

- **Index:** The draw widget's associated `IDLgrWindow` object uses indexed color. This is the default value.
- **RGB:** The RGB color model is used.

In the generated \*.pro file, this value is specified by the `COLOR_MODEL` keyword to the widget creation routine.

For information on using indexed color in Object Graphics window objects, see [Chapter 20, “Working with Color”](#) in the *Using IDL* manual.

### Colors

The Colors attribute specifies the number of colors that the drawable should attempt to use from the system color table. This property is only valid with the [Graphics Type](#) property is set to Direct, for IDL Direct Graphics. By default, the Color attribute is set

to 0, which indicates that IDL will attempt to get all available colors. That is, all or most of the available color indices are allocated, based on the window system in use. You can set the Colors attribute to any integer, but most values will be in the range of  $-256 < n < 256$ .

This property has effect only if it is supplied when the first IDL graphics window is created. To use monochrome windows on a color display, set the Colors property to 2 for the first window. One color table is maintained for all running IDL windows.

In the generated \*.pro file, this value is specified by the COLORS keyword to the widget creation routine.

## Graphics Type

The Graphics Type attribute specifies the type of graphics that the draw widget will support. These are the possible values:

- **Direct:** The draw widget will display Direct Graphics. This is the default value. The Colors property is used only when [Graphics Type](#) is set to Direct.
- **Object:** The draw widget will display IDL Object Graphics. The [Color Model](#) and [Renderer](#) properties are used only when the Graphics Type is set to Object.

In the generated \*.pro file, this value is specified with the GRAPHICS\_LEVEL keyword to the widget creation routine.

## Renderer

The Renderer attribute specifies which graphics renderer to use with IDL Object Graphics. That is, for this property to be used, the [Graphics Type](#) property should be set to Object. These are the possible values for the Renderer attribute:

- **OpenGL:** The platform's native OpenGL renderer is used when drawing objects within the window. If your platform does not have a native OpenGL implementation, IDL's software implementation is used as the renderer. This value is set by default.
- **Software:** IDL's software implementation is used when drawing objects within the window.

In the generated \*.pro file, this value is specified by the RENDERER keyword to the widget creation routine.

For more information, see "[Hardware vs. Software Rendering](#)" in Chapter 28 of the *Using IDL* manual.

**Note**

The renderer selection can also affect the maximum size of a draw widget.

**Retain**

The Retain attribute specifies how backing store is performed in the draw area. These are the possible values:

- None: There is no backing store. When the Retain property is set to None, you should track [OnExpose](#) events so that you can handle the redrawing of the screen. This is the default value.
- System: The server or window system should provide backing store.
- IDL Pixmap: IDL should provide backing store.

In the generated \*.pro file, this value is specified with the RETAIN keyword to the widget creation routine.

For information on the use of the Retain property with Direct Graphics, see “[Backing Store](#)” in Appendix B of the *IDL Reference Guide*. For more information on this property with IDL Object Graphics, see [IDLgrWindow::Init](#) in the *IDL Reference Guide*.

**Scroll**

The Scroll attribute specifies if the draw area widget will support scrolling, and will have scroll bars. By default, this value is set to False, which indicates there are no scroll bars. To display scroll bars, and enable scrolling, set this value to True. If you do so, set the size of the scrollable area with the [X Scroll](#) and [Y Scroll](#) properties.

In the generated \*.pro file, this value is specified with the SCROLL keyword to the widget creation routine.

**X Scroll**

The X Scroll attribute specifies the width in pixels of the drawing area. This width includes the exposed and virtual area. By default, this value is not set. You can set X Scroll to any width from 0 to *n*. If you set this value, also set the [Scroll](#) and [Y Scroll](#) property values.

In the generated \*.pro file, this value is specified with the XSIZE keyword to the widget creation routine.



**Note**


---

To set the width of the displayed widget, use the [X Size](#) common property.

---

**Y Scroll**

The Y Scroll attribute specifies the height in pixels of the drawing area. This height includes the exposed and virtual area. By default, this value is not set. You can set Y Scroll to any height in pixels from 0 to  $n$ . If you set this value, also set the [Scroll](#) and [X Scroll](#) properties.

In the generated \*.pro file, this value is specified with the YSIZE keyword to the widget creation routine.

**Note**


---

To set the height of the displayed widget, use the [Y Size](#) common property.

---

## Draw Area Widget Events

For draw area widgets, you can set common event properties and draw area-specific event properties. By default, event values are *not* set. For a list of events common to all widgets, see “[Common Events](#)” on page 493.

These are the draw area event properties:

**OnButton**

The OnButton value is the routine that is called when a mouse button event is detected. In the generated \*\_eventcb.pro file, the calling sequence looks like this:

```
PRO <RoutineName>, Event
```

*RoutineName* is the name of the event procedure you specify. *Event* is the returned event structure, which is of the following type:

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

Note that this is the same event structure returned for all draw area events; [OnButton](#), [OnExpose](#), [OnMotion](#), and [OnViewportMoved](#) events all return the same structure. Therefore the following paragraphs describe all these events.

TYPE returns a value that describes the type of draw widget interaction that generated an event. If there is a button press, it returns 0, and if there is a button release, it returns 1. If there is motion, it returns 2 (for an [OnMotion](#) event). If the

viewport moved with the scroll bars, it returns 3 (for an [OnViewportMoved](#) event). If the visibility changes, it returns 4 (for an [OnExpose](#) event).

The X and Y fields give the device coordinates at which the event occurred, measured from the lower left corner of the drawing area.

PRESS and RELEASE are bitmasks in which the least significant bit represents the left-most mouse button. The corresponding bit of PRESS is set when a mouse button is pressed, and in RELEASE when the button is released. If the event is a motion event, both PRESS and RELEASE returns zero.

CLICKS returns either 1 or 2. If the time interval between button-press events is greater than the time interval for a double-click event for the system, the CLICKS field returns 1. If the time interval between two button-press events is less than the time interval for a double-click event for the platform, the CLICKS field returns 2.

## OnExpose

The OnExpose value is the routine that is called when the visibility of any portion of the draw window (or viewport) changes or is exposed. In the generated \*\_eventcb.pro file, the calling sequence looks like this:

```
PRO <RoutineName>, Event
```

*RoutineName* is the name of the event procedure you specify. Event is the returned event structure, which is of the following type:

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

Note that this is the same event structure returned for all draw area events; [OnButton](#), [OnExpose](#), [OnMotion](#), and [OnViewportMoved](#) events all return the same structure. For information on this structure, see [OnButton](#).

## OnMotion

The OnMotion value is the routine that is called when a mouse motion event is detected. In the generated \*\_eventcb.pro file, the calling sequence looks like this:

```
PRO <RoutineName>, Event
```

*RoutineName* is the name of the event procedure you specify. Event is the returned event structure, which is of the following type:

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

Note that this is the same event structure returned for all draw area events; [OnButton](#), [OnExpose](#), [OnMotion](#), and [OnViewportMoved](#) events all return the same structure. For information on this structure, see [OnButton](#).

## OnViewportMoved

The `OnViewportMoved` value is the routine that is called when the viewport of a scrolling draw widget is moved, using the scroll bars. In the generated `*_eventcb.pro` file, the calling sequence looks like this:

```
PRO <RoutineName>, Event
```

*RoutineName* is the name of the event procedure you specify. `Event` is the returned event structure, which is of the following type:

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

Note that this is the same event structure returned for all draw area events; [OnButton](#), [OnExpose](#), [OnMotion](#), and [OnViewportMoved](#) events all return the same structure. For information on this structure, see [OnButton](#).

# Table Widget Properties

Table widgets display data and allow for data editing by the user. Tables can have one or more rows and one or more columns.

When programming in IDL, you create table widgets using the `WIDGET_TABLE` function. For more information, see [WIDGET\\_TABLE](#) in the *IDL Reference Guide*.

## Table Widget Attributes

For table widgets, you can set common attributes and table-specific attributes. For a list of common attributes, see “[Common Attributes](#)” on page 490. These are the table widget-specific attributes:

### Alignment

The Alignment attribute specifies how the text is aligned in the cells. These are the possible values:

- Left: The text is left-justified. This is the default value.
- Right: The text is right-justified.
- Center: The text is centered.

In the generated \*.pro file, this value is specified with the `ALIGNMENT` keyword to the widget creation routine.

### Column Labels

The Column Labels attribute specifies the labels for the table columns. By default, this value is set to empty strings, but you can set it to any set of strings. To set the labels for table rows, use the [Row Labels](#) property.

To enter more than one string in the Value field:

Type in a string, then press Control+Enter (at the end of each line). This action moves you to the next line, or the next label for a column. When you have entered as many labels as you want, press Enter to set the values.

In the generated \*.pro file, this value is specified with the `COLUMN_LABELS` keyword to the widget creation routine.

### Display Headers

The Display Headers attribute determines if the table headings, the row and column labels, are displayed. By default, this value is set to True, indicating that table

heading should be displayed. To disable the display of table headings, set this value to False.

In the generated \*.pro file, the False value is specified with the NO\_HEADERS keyword to the widget creation routine.

### **Editable**

The Editable attribute determines if the table widget is editable or not. By default, this value is set to False, which means the text widget is not editable, and the text is read-only. To make the text widget editable, set this value to True.

In the generated \*.pro file, this value is specified with the EDITABLE keyword to the widget creation routine.

### **Number of Columns**

The Number of Columns attribute specifies the number of columns in the table widget. This value sets the full, virtual width of the table. By default, it is set to 6.

In the generated \*.pro file, this value is specified with the XSIZE keyword to the widget creation routine.

### **Note**

---

To have a scrollable table, set the [Scroll](#) property to True. Then, to specify the visible size of the table, set the [Viewport Columns](#) property.

---

### **Number of Rows**

The Number of Rows attribute specifies the number of rows in the table widget. This value sets the full, virtual height of the table. By default, it is set to 6.

In the generated \*.pro file, this value is specified with the YSIZE keyword to the widget creation routine.

### **Note**

---

To have a scrollable table, set the [Scroll](#) property to True. Then, to specify the visible size of the table, set the [Viewport Columns](#) property.

---

### **Resize Columns**

The Resize Columns attribute determines if this user can resize table columns. By default, this value is set to True, indicating that the user can resize the columns. To

specify that the columns of the table are not resizeable by the user, set this value to False.

In the generated \*.pro file, this value is specified with the `RESIZEABLE_COLUMNS` keyword to the widget creation routine.

---

**Note**

If you set the [Display Headers](#) property to False, the ability to resize the columns is automatically disabled.

---

## Row/Column Major

The Row/Column Major attribute specifies how data is transferred to the table widget, either by Row or by Column. By default, this value is set to Row, indicating that the data should be read into the table as if each element of the vector is a structure containing one row's data. To specify that the data should be read into the table as if each element of the vector is a structure containing one column's data, set this value to Column. Note that for either setting to work properly the structures must all be of the same type, and must have one field for each column or row in the table.

In the generated \*.pro file, this value is specified with the `ROW_MAJOR` or the `COLUMN_MAJOR` keyword to the widget creation routine.

## Row Labels

The Row Labels attribute specifies the labels for the table rows. By default, this value is set to empty strings, but you can set it to any set of strings. To set the labels for table columns, use the [Column Labels](#) property.

To enter more than one string in the Value field:

Type in a string, then press Control+Enter (at the end of each line). This action moves you to the next line, or the next label for a row. When you have entered as many labels as you want, press Enter to set the values.

In the generated \*.pro file, this value is specified with the `ROW_LABELS` keyword to the widget creation routine.

## Scroll

The Scroll attribute determines if the table widget has scroll bars. By default, this value is set to False, indicating that the table will have no scroll bars. To enable scroll bars, set this value to True. If you set this value to True, you can set the size of the scrollable region with the [Viewport Rows](#) and [Viewport Columns](#) properties.

In the generated \*.pro file, this value is specified with the SCROLL keyword to the widget creation routine.

## Viewport Columns

The Viewport Columns attribute specifies the number of columns that should be visible in the scroll area of the table widget. By default, this value is set to 6.

If you first set the [Scroll](#) property to True, you can then set this value to any size from 0 to  $n$  columns within the limits of your full table size. The full table size, or virtual width in columns, is set with the [Number of Columns](#) property.

This property is used only when the [Component Sizing](#) property is set to Default. If you set the Component Sizing property to Explicit, either through the Properties dialog or by dragging the component to specific size, the Viewport Columns property is ignored, and the [X Size](#) and the [Y Size](#) properties are used.

In the generated \*.pro file, this value is specified with the X\_SCROLL\_SIZE keyword to the widget creation routine.

## Viewport Rows

The Viewport Rows attribute specifies the number of rows that should be visible in the scroll area of the table widget. By default, this value is set to 6.

If you first set the [Scroll](#) property to True, you can then set this value to any size from 0 to  $n$  rows, within the limits of your full table size. The full table size, or virtual height in rows, is set with the [Number of Rows](#) property.

This property is used only when the [Component Sizing](#) property is set to Default. If you set the Component Sizing property to Explicit, either through the Properties dialog or by dragging the component to specific size, the Viewport Rows property is ignored, and the [X Size](#) and the [Y Size](#) properties are used.

In the generated \*.pro file, this value is specified with the Y\_SCROLL\_SIZE keyword to the widget creation routine.

## Table Widget Events

For table widgets, you can set common event properties and table-specific event properties. By default, event values are *not* set. For a list of events common to all widgets, see [“Common Events”](#) on page 493.

These are the table widget-specific event properties:

## OnCellSelect

The OnCellSelect value is the routine that is called when cells are selected in the table. When you set this value, the calling sequence looks like this in the generated \*\_eventcb.pro file:

```
PRO <RoutineName>, Event
```

*RoutineName* is the name of the event procedure you specify. *Event* is the returned event structure, which is returned when range of cells is selected or deselected and is of the following type:

```
{ WIDGET_TABLE_CELL_SEL, ID:0L, TOP:0L, HANDLER:0L, TYPE:4,
  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. When cells are deselected, either by changing the selection or by clicking in the upper left corner of the table, an event is generated in which the SEL\_LEFT, SEL\_TOP, SEL\_RIGHT, and SEL\_BOTTOM fields contain the value -1.

---

### Note

Two WIDGET\_TABLE\_CELL\_SEL events are generated when an existing selection is changed to a new selection. If your code uses this event, be sure to differentiate between select and deselect events.

---

## OnColWidth

The OnColWidth value is the routine that is called when the column width is changed. When you set this value, the calling sequence looks like this in the generated \*\_eventcb.pro file:

```
PRO <RoutineName>, Event
```

*RoutineName* is the name of the event procedure you specify. *Event* is the returned event structure, which is returned when a column width is changed by the user and is of the following type:

```
{ WIDGET_TABLE_COLUMN_WIDTH, ID:0L, TOP:0L, HANDLER:0L, TYPE:7,
  COLUMN:0L, WIDTH:0L }
```

COLUMN contains the zero-based column number, and WIDTH contains the new width.



## OnDelete

The OnDelete value is the routine that is called when text is deleted from the table. When you set this value, the calling sequence looks like this in the generated \*\_eventcb.pro file:

```
PRO <RoutineName>, Event
```

*RoutineName* is the name of the event procedure you specify. *Event* is the returned event structure, which is returned when any amount of text is deleted from a cell of a table widget and is of the following type:

```
{ WIDGET_TABLE_DEL, ID:0L, TOP:0L, HANDLER:0L, TYPE:2, OFFSET:0L,
  LENGTH:0L, X:0L, Y:0L }
```

OFFSET is the (zero-based) character position of the first character deleted, and it is the insertion position that will result when the next character is inserted. LENGTH gives the number of characters involved. The X and Y fields give the zero-based address of the cell within the table.

## OnFocus

The OnFocus value is the routine that is called when the keyboard focus of the base changes. When you set it, the calling sequence looks like this in the generated \*\_eventcb.pro file:

```
PRO <RoutineName>, Event
```

*RoutineName* is the name of the event procedure you specify. *Event* is the returned event structure, which is of the following type:

```
{ WIDGET_KBRD_FOCUS, ID:0L, TOP:0L, HANDLER:0L, ENTER:0 }
```

ENTER returns 1 (one) if the table widget is gaining the keyboard focus, or 0 (zero) if the table widget is losing the keyboard focus.

## OnInsertChar

The OnInsertChar value is the routine that is called when text is inserted in the table. When you set this value, the calling sequence looks like this in the generated \*\_eventcb.pro file:

```
PRO <RoutineName>, Event
```

*RoutineName* is the name of the event procedure you specify. *Event* is the returned event structure, which is returned when a single character is typed into a cell of a table widget and is of the following type:

```
{ WIDGET_TABLE_CH, ID:0L, TOP:0L, HANDLER:0L, TYPE:0, OFFSET:0L,
  CH:0B, X:0L, Y:0L }
```

OFFSET is the (zero-based) insertion position that will result after the character is inserted. CH is the ASCII value of the character. The X and Y fields indicate the zero-based address of the cell within the table.

### OnInsertString

The OnInsertString value is the routine that is called when text is inserted in the table. When you set this value, the calling sequence looks like this in the generated \*\_eventcb.pro file:

```
PRO <RoutineName>, Event
```

*RoutineName* is the name of the event procedure you specify. Event is the returned event structure, which is returned when multiple characters are pasted into a cell and is of the following type:

```
{ WIDGET_TABLE_STR, ID:0L, TOP:0L, HANDLER:0L, TYPE:1, OFFSET:0L,
  STR:'', X:0L, Y:0L }
```

OFFSET is the (zero-based) insertion position that will result after the text is inserted. STR is the string to be inserted. The X and Y fields indicate the zero-based address of the cell within the table.

### OnInvalidData

The OnInvalidData value is the routine that is called when invalid data is set in a cell. When you set this value, the calling sequence looks like this in the generated \*\_eventcb.pro file:

```
PRO <RoutineName>, Event
```

*RoutineName* is the name of the event procedure you specify. Event is the returned event structure, which is returned when the text entered by the user does not pass validation, and the user has finished editing the field (by pressing Tab or Enter). The event structure is of the following type:

```
{ WIDGET_TABLE_INVALID_ENTRY, ID:0L, TOP:0L, HANDLER:0L, TYPE:8,
  STR:'', X:0L, Y:0L }
```

STR contains invalid contents entered by the user as a text string. The X and Y fields contain the cell location.

### OnTextSelect

The OnTextSelect value is the routine that is called when text is selected in the table. When you set this value, the calling sequence looks like this in the generated \*\_eventcb.pro file:

```
PRO <RoutineName>, Event
```

*RoutineName* is the name of the event procedure you specify. *Event* is the returned event structure, which is returned when an area of text is selected. The event structure is of the following type:

```
{WIDGET_TABLE_TEXT_SEL, ID:0L, TOP:0L, HANDLER:0L, TYPE:3,  
  OFFSET:0L, LENGTH:0L, X:0L, Y:0L}
```

This event announces a change in the insertion point. *OFFSET* is the (zero-based) character position of the first character to be selected. *LENGTH* gives the number of characters involved. A *LENGTH* of zero indicates that the widget has no selection, and that the insertion position is given by *OFFSET*. The *X* and *Y* fields indicate the zero-based address of the cell within the table.





# Chapter 18: Widgets

The following topics are covered in this chapter:

---

Overview .....	542	Using Draw Widgets .....	571
Widget Types .....	544	Creating Menus .....	573
Manipulating Widgets .....	549	Controlling Widgets .....	578
Examples of Widget Programming .....	550	Widget Example 3 .....	581
The Widget Application Model .....	551	Widget Sizing .....	583
Creating Widget Applications .....	554	Event Processing And Callbacks .....	589
Widget Example 1 .....	557	Managing Widget Application State .....	592
Widget Values .....	559	Compound Widgets .....	594
Widget User Values .....	562	Tips on Creating Widget Applications .....	596
Widget Events .....	563	Compound Widget Example .....	598
Widget Example 2 .....	569		

# Overview

IDL allows you to construct and manipulate graphical user interfaces using *widgets*. Widgets (or *controls*, in the terminology of some development environments) are simple graphical objects such as pushbuttons or sliders that allow user interaction via a pointing device (usually a mouse) and a keyboard. This style of graphical user interaction offers many significant advantages over traditional command-line based systems.

IDL widgets are significantly easier to use than other alternatives, such as writing a C language program using the native window system directly. IDL handles much of the low-level work involved in using such toolkits. The interpretive nature of IDL makes it easy to prototype potential user interfaces. In addition to the user interface, the author of a program written in a traditional compiled language also must implement any computational and graphical code required by the program. IDL widget programs can draw on the full computational and graphical abilities of IDL to supply these components.

The style of widgets IDL creates depends on the windowing system supported by your host computer. Unix and VMS hosts use Motif widgets, while Microsoft Windows and Macintosh systems use their native toolkits. Although the different toolkits produce applications with a slightly different look and feel, most properly-written widget applications work on all systems without change.

IDL graphical user interfaces are constructed by combining widgets in a treelike hierarchy. Each widget has one parent widget and zero or more child widgets. There is one exception: the topmost widget (called a *top-level base*) is always a base widget and has no parent.

Programs that use widgets are *event driven*. In an event driven system, the program creates an interface and then waits for messages (events) to be sent to it from the window system. Events are generated in response to user manipulation, such as pressing a button or moving a slider. The program responds to events by carrying out the action or computation specified by the programmer, and then waiting for the next event. This approach to computing is fundamentally different from the traditional command-based approach.

---

**Note**

You can use the IDL GUIBuilder to create user interfaces interactively. The IDL GUIBuilder allows you to create and interface rapidly and generate the IDL source

code to create the interface. For information, see [Chapter 17, “Using the IDL GUIBuilder”](#).

---

## Running the Example Code

The example code used in this chapter is part of the IDL distribution. All of the files mentioned are located in the `doc` subdirectory of the `examples` subdirectory of the main IDL directory. By default, this directory is part of IDL's path; if you have not changed your path, you will be able to run the examples as described here. See [!PATH](#) in the *IDL Reference Guide* for information on IDL's path.

# Widget Types

IDL supports two types of widgets. *Widget primitives* are the base interface elements. *Compound widgets* are more complex interface elements built in the IDL language from the widget primitives. In addition, there are a number of *dialogs* which are widget-like but which do not belong to a widget hierarchy.

## Widget Primitives

Widget primitives are created by functions with names like `WIDGET_BASE` and `WIDGET_BUTTON`. IDL provides the following widget primitives:

### Base

A base is a widget used to hold other widgets, including other base widgets. Base widgets can optionally contain scroll bars that allow the base to be larger than the space on the screen. In this case, only part of the base is visible at any given time, and the scroll bars are used to control which part is visible.

Base widgets are created by the `WIDGET_BASE` function. See [WIDGET\\_BASE](#) in the *IDL Reference Guide* for more information.

*Top-level bases* are a special class of base widget created without a parent widget ID. Top-level bases can be organized into an application hierarchy by specifying the `GROUP_LEADER` keyword. Top-level bases can be made to *float* above their group leaders (via the `FLOATING` keyword), or can be created as *modal* bases (via the `MODAL` keyword) that interrupt program execution until the user performs some action. See “[The Widget Application Model](#)” on page 551 for additional discussion of widget applications.

### Button

A pushbutton is activated by moving the mouse cursor over the button and pressing a mouse button. Button widgets are created by the `WIDGET_BUTTON` function. See [WIDGET\\_BUTTON](#) in the *IDL Reference Guide* for more information.

### Draw

Draw widgets offer a rectangular area that works like a standard IDL graphics window. Draw widgets can use either Direct graphics or Object graphics, depending on how they are created. Any graphical output that can be produced by IDL can be directed to one of these widgets, either through the [WSET](#) function or by using the object reference of a draw widget’s `IDLgrWindow` object. Draw widgets can optionally contain scrollbars that allow examining a graphical region larger than the



area containing the widget. Draw widgets are created by the `WIDGET_DRAW` function. See [WIDGET\\_DRAW](#) in the *IDL Reference Guide* for more information.

## Droplist

Droplist widgets display a single entry from a list of options. When selected, they reveal the entire list. When a new option is selected from this list, the list disappears and the new selection is displayed. On systems using the Motif window system, Droplist widgets look like buttons with labels that change depending on the item selected from the drop-down list. Droplist widgets are created by the `WIDGET_DROPLIST` function. See [WIDGET\\_DROPLIST](#) in the *IDL Reference Guide* for more information.

## Label

Label widgets display static text. They are similar to single-line text widgets but are optimized for small labeling purposes. Text widgets should be used to display large amounts of text. Label widgets are created by the `WIDGET_LABEL` function. See [WIDGET\\_LABEL](#) in the *IDL Reference Guide* for more information.

## List

A list widget offers the user a list of text elements from which to choose. Users can select an item by pointing with the mouse cursor and pressing a button. List widgets always have a vertical scrollbar and allow selection from a large number of items. List widgets are created by the `WIDGET_LIST` function. See [WIDGET\\_LIST](#) in the *IDL Reference Guide* for more information.

## Slider

Slider widgets are used to select or indicate a value within a range of possible integer values. They consist of a rectangular region that represents the possible range of values. Inside this region is a sliding pointer that displays the current value. This pointer can be manipulated by the user via the mouse or from within IDL by the `WIDGET_CONTROL` procedure. Slider widgets are created by the `WIDGET_SLIDER` function. See [WIDGET\\_SLIDER](#) in the *IDL Reference Guide* for more information.

## Table

Table widgets are used to display information in tabular format. Individual table cells (or ranges of cells) can be selected for editing by the user. Table widgets are created by the `WIDGET_TABLE` function. See [WIDGET\\_TABLE](#) in the *IDL Reference Guide* for more information.

## Text

Text widgets are used to display text and to get text input from the user. They can have one or more lines and can optionally contain scroll bars that allow viewing more text than can otherwise be displayed. Text widgets are created by the `WIDGET_TEXT` function. See [WIDGET\\_TEXT](#) in the *IDL Reference Guide* for more information.

## Compound Widgets

A compound widget is a complete, self-contained, reusable widget sub-tree that behaves to a large degree just like a widget primitive, but which is written in the IDL language. Compound widget routines can be found (along with many other routines that use the widgets) in the `lib` subdirectory of the IDL distribution. All compound widget filenames begin with “CW\_” to make them easier to identify. The following types of compound widgets are included in the IDL distribution.

### Animation

The `CW_ANIMATE` compound widget — along with its associated routines — displays an animated sequence of images. See [CW\\_ANIMATE](#) in the *IDL Reference Guide*.

### Color Manipulation

The `CW_CLR_INDEX` compound widget displays a color bar and allows the user to select a color index. See [CW\\_CLR\\_INDEX](#) in the *IDL Reference Guide*.

The `CW_COLORSEL` compound widget displays all the colors in the current colormap and allows the user to select color indices. See [CW\\_COLORSEL](#) in the *IDL Reference Guide*.

The `CW_RGBSLIDER` compound widget allows the user to adjust color values using the RGB, CMY, HSV, and HLS color systems. See [CW\\_RGBSLIDER](#) in the *IDL Reference Guide*.

### Data Entry and Display

The `CW_FIELD` compound widget simplifies building data-entry interfaces by combining label and text widgets. See [CW\\_FIELD](#) in the *IDL Reference Guide*.

The `CW_FORM` compound widget allows you to create simple forms with text, numeric fields, buttons, and droplists. See [CW\\_FORM](#) in the *IDL Reference Guide*.

## Image Manipulation

The `CW_DEFROI` compound widget allows you to specify a *region of interest* within a draw widget. See [CW\\_DEFROI](#) in the *IDL Reference Guide*.

The `CW_ZOOM` compound widget displays original and zoomed images side-by-side. See [CW\\_ZOOM](#) in the *IDL Reference Guide*.

## Orientation

The `CW_ARCBALL` compound widget allows the user to intuitively specify three-dimensional orientations. See [CW\\_ARCBALL](#) in the *IDL Reference Guide*.

The `CW_ORIENT` compound widget allows the user to interactively adjust the three-dimensional drawing transformation. See [CW\\_ORIENT](#) in the *IDL Reference Guide*.

## User Interface

The `CW_BGROUP` compound widget simplifies creation of a cluster of buttons. Button groups can be simple menus in which each button acts independently, *exclusive* groups (also known as “radio buttons”), or *non-exclusive* groups (often called “checkboxes”). See [CW\\_BGROUP](#) in the *IDL Reference Guide*.

The `CW_FSLIDER` compound widget is a version of the slider widget that handles floating-point values. See [CW\\_FSLIDER](#) in the *IDL Reference Guide*.

The `CW_PDMENU` compound widget creates pulldown menus, which can include sub-menus, from a set of buttons. See [CW\\_PDMENU](#) in the *IDL Reference Guide*.

See “[Writing Compound Widgets](#)” on page 594 for information on writing your own compound widgets.

## Dialogs

A dialog is a widget-like user interface element that is not part of a widget hierarchy. Dialogs are *modal* (or “blocking”) elements, which means that when a dialog is displayed, no other interface elements (widgets or compound widgets) can be manipulated until the user dismisses the dialog.

## File and Directory Selection

File selection dialogs allow you to choose a file or directory via a graphical interface. The `DIALOG_PICKFILE` function returns the string containing the name of the selected file. See [DIALOG\\_PICKFILE](#) in the *IDL Reference Guide* for more information.

## Message

Message dialogs are *modal* (or “blocking”) dialog boxes that can display warnings, informational messages, or error messages. When a message dialog is displayed, no widgets can be manipulated until the user dismisses the dialog by clicking on one of its buttons. Message dialogs do not belong to widget hierarchies; they are instantly created when the `DIALOG_MESSAGE` function is called and block all widget activity until dismissed. See [DIALOG\\_MESSAGE](#) in the *IDL Reference Guide* for more information.

## Printing

IDL provides two dialogs for controlling printing. `DIALOG_PRINTJOB` opens a native dialog that allows you to set the properties of a printer, parameters for a printing job (number of copies to print, for example). `DIALOG_PRINTERSETUP` opens a native dialog for setting the applicable properties for a particular printer. See [DIALOG\\_PRINTJOB](#) and [DIALOG\\_PRINTERSETUP](#) in the *IDL Reference Guide* for more information.

# Manipulating Widgets

Widgets are controlled via their *widget IDs*. The widget ID is a long integer assigned to the widget when it is first created. In practice, the widget ID of a widget is contained in a named variable that you assign when you call the widget creation function. For example, you might create a base widget with the following IDL command:

```
base = WIDGET_BASE()
```

Here, the IDL variable `base` contains the widget ID of the top-level widget base that is created.

IDL provides several routines that allow you to manipulate and manage widgets:

- **WIDGET\_CONTROL** allows you to *realize* (make visible on your screen) widget hierarchies, manipulate them, and destroy them when you are finished.
- **WIDGET\_EVENT** allows you to process events generated by a specific widget hierarchy.
- **WIDGET\_INFO** allows you to obtain information about the state of a specific widget or widget hierarchy.
- **XMANAGER** provides an event loop and manages events generated by a widget hierarchy.
- **XREGISTERED** allows you to test whether a specific widget is currently registered with XMANAGER.

These widget manipulation routines are discussed in more detail in the following sections.

## Examples of Widget Programming

A number of simple examples of widget programming can be seen by running the IDL program `examples.pro`, which can be found in the `/examples/misc` folder of the IDL distribution. A widget interface with a pulldown menu of small widget applications should appear.

# The Widget Application Model

Using widgets, you can create entire IDL applications with graphical user interfaces. Although widget applications are running “inside” IDL, a well-designed program can behave and appear just like a stand-alone application.

A widget application consists of a *group* of top-level bases organized hierarchically. Groups of widgets are defined by setting the `GROUP_LEADER` keyword when creating the widget. Group membership controls how and when widgets are iconized, which layer they appear in, and when they are destroyed.

The following figure depicts a widget application group hierarchy consisting of six top-level bases in three groups: base 1 leads all six bases, base 2 leads bases 4 and 5, and base 3 leads base 6. What does this mean? Operations that affect base 2 also affect bases 4 and 5. Operations that affect base 3 also affect base 6. Operations that affect base 1 affect all six bases—that is, a group includes not only those bases that explicitly claim one base as their leader, but also all bases *led* by those member bases.

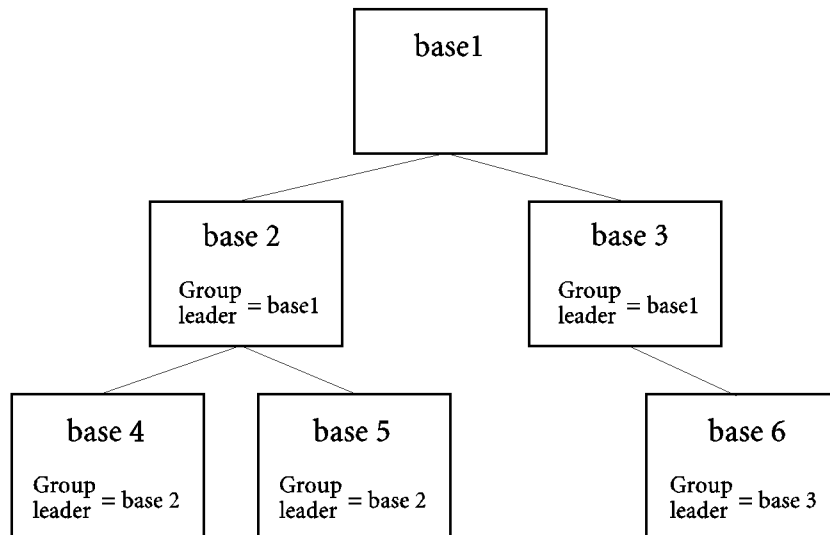


Figure 18-1: A widget application group hierarchy with six top-level bases.

The following IDL commands would create this hierarchy:

```

base1 = WIDGET_BASE()
base2 = WIDGET_BASE(GROUP_LEADER=base1)

```

```

base3 = WIDGET_BASE (GROUP_LEADER=base1)
base4 = WIDGET_BASE (GROUP_LEADER=base2)
base5 = WIDGET_BASE (GROUP_LEADER=base2)
base6 = WIDGET_BASE (GROUP_LEADER=base3)

```

## Iconization

On Motif and Windows platforms, bases and groups of bases can be *iconized* (or *minimized*) by clicking the system minimize control. When a group leader is iconized, all members of the group are minimized as well. Minimization has no meaning on the Macintosh.

## Layering

*Layering* is the process by which groups of widgets seem to share the same plane on the display screen. Within a layer on the screen, widgets have a *Z-order*, or front-to-back order, that defines which widgets appear to be on top of other widgets.

All widgets within a group hierarchy share the same layer—that is, when one group member has the input focus, all members of the group hierarchy are displayed in a layer that appears in front of all other groups or applications. Within the layer, the widgets can have an arbitrary Z-order.

## Destruction

When a group leader widget is destroyed, either programmatically or by clicking on the system “close” button, all members of the group are destroyed as well.

See [Iconizing, Layering, and Destroying Groups of Top-Level Bases](#) under `WIDGET_BASE` in the *IDL Reference Guide* for detailed information on how group membership defines widget behavior on different platforms.

## Floating bases

Top-level base widgets created with the `FLOATING` keyword set will *float* above their group leaders, even though they share the same layer. Floating bases and their group leaders are iconized in a single icon (on platforms where iconization is possible). Floating bases are destroyed when their group leaders are destroyed.

## Modal bases

Top-level base widgets created with the `MODAL` keyword will float above their group leaders, and will suspend processing in the widget application until they are dismissed. (*Dialogs* are generally modal.) Modal bases cannot be iconized, and on some platforms other bases cannot be moved or iconized while the modal dialog is present. Modal bases cannot have scroll bars or menubars.



## Menubars

Widget applications can have an application-specific menubar, created by the `APP_MBAR` keyword to `WIDGET_BASE`. Currently, application menubars are equivalent to individual menubars created by the `MBAR` keyword on Motif and Windows platforms. On the Macintosh, the menubar defined by `APP_MBAR` “takes over” the Macintosh system menubar, while menubars defined by `MBAR` are included on an individual top-level base widget.

# Creating Widget Applications

An application using widgets goes through the following cycle:

## Construct the Widget Hierarchy

You must first build a widget hierarchy. Start with one or more *top-level bases* (created with the `WIDGET_BASE` function) in a hierarchy described by `GROUP_LEADER` relationships. Combine other widget creation functions — `WIDGET_BUTTON`, `CW_PDMENU`, etc. — to create and organize the user interface of your widget application. At this point, the widgets exist only within IDL—nothing has been created or displayed on the window system.

## Provide an Event-Handling Routine

In order for a widget application to *do* anything, you must provide a routine that examines events, determines what action to take, and implements the action. Actions may involve computation, graphics display, or updating the widget interface itself.

For the best performance, it is important that the program spend most of its time in the *event loop* provided by the event handling routine. Some widgets will not respond rapidly to user manipulation when not in this loop. Widget-based programs should wait for user-generated events, handle them as quickly as possible, and quickly return to wait for more events. Event processing is discussed in detail in “[Widget Events](#)” on page 563 and in “[Event Processing And Callbacks](#)” on page 589.

Event handling routines can use the `WIDGET_CONTROL` procedure to manipulate widgets. Possible actions include the following:

- Obtain or change the value of a widget (see “[Widget Values](#)” on page 559) using the `APPEND`, `GET_VALUE`, and `SET_VALUE` keywords.
- Obtain or change the value of a widget’s user value (discussed in “[Widget User Values](#)” on page 562) using the `GET_UVALUE` and `SET_UVALUE` keywords.
- Map and unmap widgets using the `MAP` keyword. Unmapped widgets are removed from the screen and become invisible, but they still exist.
- Change a widget’s sensitivity using the `SENSITIVE` keyword. When a widget is insensitive, it indicates the fact by changing its appearance (often by graying itself or displaying text with dashed lines) and ignores any user input. It is useful to make widgets insensitive at points where it would be inconvenient to get events from them (for example, if your program is waiting for input from another source).

- Change the settings of toggle buttons using the `SET_BUTTON` keyword.
- Push a widget hierarchy behind the other windows on the screen, or pull them in front using the `SHOW` keyword.
- If you expect an operation to be slow, display the “hourglass” cursor while the application is busy and not able to respond to user actions by setting the `HOURGLASS` keyword.

## Realize the Widgets

Bring the widget hierarchy into existence using the `REALIZE` keyword to the `WIDGET_CONTROL` procedure. This causes the widgets to be created and displayed.

## Register the Program with the XMANAGER

Register the program with the `XMANAGER` procedure. Your widget application then waits for events to be reported to it and reacts as specified in the event handling routine.

Events are obtained by `XMANAGER` via the `WIDGET_EVENT` function and passed to the calling routine (your event handler) in the form of an IDL structure variable. Each type of widget returns a different type of structure, the exact form of which is described in the documentation for the individual widget creation functions in the *IDL Reference Guide*. However, every event structure has the same first three elements. These are long integers named `ID`, `TOP`, and `HANDLER`. `ID` is the widget ID of the widget generating the event. `TOP` is the widget ID of the top-level base containing `ID`. `HANDLER` is important for event handler functions, which are discussed later in this chapter.

When an event appears, `XMANAGER` passes it to an event-handling procedure specified by the program, and the event handler takes some appropriate action based on the event. This means that multiple widget applications can run simultaneously — `XMANAGER` dispatches the events to the appropriate routine.

## Destroy the Widgets

When the application has finished (usually when the user clicks on a “Done” or “Quit” button), destroy the widget hierarchy using the `WIDGET_CONTROL` procedure’s `DESTROY` keyword. This causes all resources related to the hierarchy to be freed and removes it from the screen.

## Handling Widget Application Errors

At times, widget applications may experience errors that stop the processing of widget events by XMANAGER. This is most common during the development of the application, when unexpected programming errors are likely to appear.

By default, XMANAGER catches errors and continues processing (see [CATCH](#) in the *IDL Reference Guide*). If you are using XMANAGER to manage your widget application (as in most cases you should), have explicitly set CATCH=0, and the widget stops responding, error messages appear in your command log, and the IDL command prompt reappears, do the following:

1. Enter RETALL at the IDL prompt to return to the main program level.
2. Enter XMANAGER at the IDL prompt. Calling XMANAGER with no parameters tells it to skip registering a new application and to simply resume event processing.

---

**Note**

If you do *not* restart XMANAGER, widget applications will not respond even if you recompile.

---

# Widget Example 1

The following example demonstrates the simplicity of widget programming. It creates a base widget containing a single button, labelled “Done.” When you position the mouse cursor over the button and click, the widget is destroyed.

Enter the two procedures listed below — either in a text file named `widget1.pro`, or directly into IDL using the `.RUN` command. Enter `widget1` at the IDL prompt to run the program.

```
PRO widget1_event, ev
IF ev.select THEN WIDGET_CONTROL, ev.top, /DESTROY
END

PRO widget1
base = WIDGET_BASE()
button = WIDGET_BUTTON(base, value='Done')
WIDGET_CONTROL, base, /REALIZE
XMANAGER, 'Widget1', base
END
```

Alternately, you can run the program from the IDL distribution by entering:

```
widget1
```

at the IDL command prompt. See [“Running the Example Code”](#) on page 543 if IDL does not run the program as expected.

While this simple example does nothing particularly useful, it does illustrate some basic concepts of event-driven programming. Let’s examine how the example is constructed.

First, note that the “application” consists of two parts; an event handling routine and a creation routine. First, let’s examine the second part — the creation routine — contained in the `widget1` procedure.

The `widget1` procedure does the following:

1. Creates a top-level base widget named `base`. All widget applications have at least one base.
2. Creates a button widget named `button` with `base` as its parent. The value “Done” is assigned to the button. The value of a button widget is the text that appears on the button’s face.

3. Realizes the widget hierarchy built on `base` by calling `WIDGET_CONTROL` with the `/REALIZE` keyword. This causes the widget to appear on your computer screen.
4. Invokes the `XMANAGER` routine to manage the widget event loop, providing the name of the calling routine (`widget1`) and the name of the top-level base the widget hierarchy is built on (`base`).

The first part, contained in the `widget1_event` procedure, is the event handling routine for the application. By convention, the `XMANAGER` procedure looks for an event handling procedure with the same name as the procedure that creates the widgets, with “\_event” appended to the end. (This default can be overridden by specifying an event handler directly using the `EVENT_HANDLER` keyword to `XMANAGER`.) When an event is received by `XMANAGER`, the event structure is passed to the `widget1_event` procedure via the `ev` argument.

In this example, all the event handling routine does is check to see if the event passed to it was a `select` event, which is part of the event structure generated by the button widget. If a `select` event is received, the routine calls `WIDGET_CONTROL` with the `DESTROY` keyword to destroy the widget hierarchy built on the top-level base widget that contains the button widget (specified in the `top` field of the event structure).

For further discussion of widget events and event structures, see “[Widget Events](#)” on page 563. For details about the event structures returned by different widgets, see the documentation for each widget in the *IDL Reference Guide*.

# Widget Values

Many widget primitives have values associated with them. Initial values are set using the `VALUE` keyword to the widget creation function. The value can be obtained and/or changed at any time using the `GET_VALUE` and `SET_VALUE` keywords to the `WIDGET_CONTROL` procedure. Widgets with a value are listed below.

## Button

Type: Scalar string (text) or byte array (bitmap). The value is the button label. The `GET_VALUE` keyword cannot be used to obtain bitmaps.

To specify text as a button label, use the `VALUE` keyword as follows:

```
button = WIDGET_BUTTON(base, VALUE='Text')
```

To specify a bitmap as a button label, use the `VALUE` keyword as follows:

```
button = WIDGET_BUTTON(base, VALUE='mybitmap.bmp', /BITMAP)
```

Note that when specifying a bitmap as a button label, you must use the `BITMAP` keyword.

## Draw

Type: Integer. For draw widgets created using Direct Graphics, the value is the IDL window number for use with direct graphics routines, such as `WSET`. For draw widgets created using Object Graphics, the value is the object reference to the `IDLgrWindow` object. This value cannot be set or modified. See “[Using Draw Widgets](#)” on page 571.

## Label

Type: Scalar string. The value is the label text.

## List or Droplist

Type: Scalar string or string array. The value represents the list elements. This value can only be set; it cannot be retrieved.

## Slider

Type: Integer. The value is the current slider position.

## Table

Type: Any data type or types, organized either in a two-dimensional array or as a vector of structures. The value is the contents of the table.

If the value is specified as a two-dimensional array, all data must be of the same type.

If the value is specified as a vector of structures, all of the structures must have the same structure definition. Individual fields within the structures can be of any data type. The structures must contain one field for each column (if the `COLUMN_MAJOR` keyword to `WIDGET_TABLE` is set) or one field for each row (if the `ROW_MAJOR` keyword to `WIDGET_TABLE` is set, or if neither keyword is set).

### **Text**

Type: Scalar string or string array. The value is the contents of the text widget. When setting this value, the `APPEND` keyword to `WIDGET_CONTROL` causes the new text to be appended to the old text instead of replacing it.

## **Widget Values of Compound Widgets**

Many compound widgets also have associated values. Initial values can often be specified using the `VALUE` keyword to the creation routine. Note, however, that in some cases widget values of compound widgets cannot be set until after the widget is realized; values are thus set, obtained, or changed using the `GET_VALUE` and `SET_VALUE` keywords to the `WIDGET_CONTROL` procedure. See the documentation for the individual compound widget creation routines in the *IDL Reference Guide* for more detailed information. Compound widgets with a value are listed below:

### **CW\_ARCBALL**

Type: 3 by 3 array. The value is the three-dimensional rotation matrix.

### **CW\_BGROUP**

Type: Integer or Vector. For “normal” button groups, there is no value. For exclusive button groups, the value is the integer index of the selected button. For non-exclusive button groups, the value is a vector indicating which buttons are selected. The initial value of a button group can be set using the `SET_VALUE` keyword to `CW_BGROUP`.

### **CW\_CLR\_INDEX**

Type: Integer. The value is the index of the color selected. The value cannot be set before the widget is realized.



**CW\_COLORSEL**

Type: Integer. The value is the index of the color selected. The value cannot be set before the widget is realized.

**CW\_FIELD**

Type: String. The value is the string value of the text portion of the field widget.

**CW\_FORM**

Type: Structure. The value is a structure of tag/value pairs for each field in the form. The value cannot be set before the widget is realized.

**CW\_FSLIDER**

Type: Floating-point number. The value is the numeric value of the slider.

**CW\_ZOOM**

Type: byte array. The value is the current array displayed. Note that you must use the SET\_VALUE keyword to WIDGET\_CONTROL to display the original image.

# Widget User Values

Every widget primitive and compound widget has the potential to carry a user-specified value of any IDL data type and organization. That is, every widget contains a variable that can store arbitrary information. This value is ignored by the widget and is for the programmer's convenience only.

The initial *user value* is specified using the UVALUE keyword to the widget creation function. If no initial value is specified, the user value is undefined. Once the widget exists, its user value can be examined and/or changed using the GET\_UVALUE and SET\_UVALUE keywords to the WIDGET\_CONTROL procedure. The widget user value should not be confused with the concept of widget values described above.

## User Values Simplify Event Handling

User values can be used to simplify event-handling. If each widget in a widget hierarchy has a distinct user value, you need only check the user value of any event to determine which widget generated it. In practice, this means you do not need to keep track of the widget IDs of all the widgets in your widget hierarchy in order to determine what to do with a given event.

## User Values Can Simulate Global Variables

Another use for user variables is to simulate a variable that is “known” in more than one IDL routine. For example, you can set the user value of a top-level base widget equal to one or more widget IDs. You then have an easy way to “import” the widget IDs from your widget creation routine into your event handling routine.

We will take advantage of both of these aspects of user values in our next example, “[Widget Example 2](#)” on page 569.

# Widget Events

The concept of events and event processing underlies every aspect of widget programming. It is important to understand how IDL handles widget events in order to use widgets effectively.

## What are Widget Events?

A widget event is a message returned from the window system as the result of user interactions such as pressing a button or otherwise manipulating a widget. In response to an event, a widget program usually performs some action for the user (e.g., opens a file, updates a plot).

## Structure of Widget Events

As events arrive from the window system, IDL saves them in a queue for the target widget. They are delivered to the IDL program as IDL structures by the `WIDGET_EVENT` function. Every widget event structure has the same first three fields: these are long integers named `ID`, `TOP`, and `HANDLER`.

- `ID` is the widget ID of the widget generating the event.
- `TOP` is the widget ID of the top-level base containing `ID`.
- `HANDLER` is the widget ID of the widget associated with the event handling routine. The importance of `HANDLER` will become apparent when we discuss event routines and compound widgets, below.

Event structures for different widgets may contain other fields as well. The exact form of the event structure for any given widget is described in the documentation for that widget's creation function in the *IDL Reference Guide*.

## Processing Widget Events

All widget event processing in IDL is handled by the `WIDGET_EVENT` function. Note that while we will discuss `WIDGET_EVENT` here for completeness, in most cases you will *not* want to call `WIDGET_EVENT` directly. The `XMANAGER` routine provides a convenient, simplified interface to `WIDGET_EVENT` and allows IDL to take over the task of managing multiple widget applications.

### Calling the `WIDGET_EVENT` Function

In its simplest form, the `WIDGET_EVENT` function is called with a widget ID (usually, the ID of a widget base) as its argument. `WIDGET_EVENT` checks the

queue of undelivered events for that widget *or any of its children*. If an event is present, it is immediately dequeued and returned. If no event is available, `WIDGET_EVENT` blocks until one arrives and then returns it. Typically, the request is made for a top-level base, so `WIDGET_EVENT` returns events for any widget in that widget hierarchy (also called a widget tree).

This simple usage suffers from a major weakness. Since each call to `WIDGET_EVENT` is looking for events from a specified widget hierarchy, it is not possible to receive events for more than one widget hierarchy at a time. It is important to be able to run multiple widget applications (each with a separate top-level base) simultaneously. An example would be an image processing application, a colorable manipulation tool, and an on-line help reader all running together.

One solution to this problem is to call `WIDGET_EVENT` with an array of widget identifiers instead of a single ID. In this case, `WIDGET_EVENT` returns events for any widget hierarchy in the list. This solution is effective, but it still requires that you maintain a complete list of all interesting top-level base identifiers, which implies that all cooperating applications need to know about each other.

The most powerful way to use `WIDGET_EVENT` is to call it without any arguments at all. Called this way, it will return events for any currently-realized widgets that have expressed an interest in being managed. (You specify that a widget wants to be managed by setting the `MANAGED` keyword to the `WIDGET_CONTROL` procedure.) This form of `WIDGET_EVENT` is especially useful when used in conjunction with widget event callback routines, discussed in [“Event Processing And Callbacks”](#) on page 589.

## Managing Events with XMANAGER

As discussed above, `WIDGET_EVENT` provides basic widget event-handling capabilities. However, it is extremely rare for a user-written widget program to actually call `WIDGET_EVENT` directly. Instead, your programs should call the `XMANAGER` procedure, which provides a convenient, simplified interface to `WIDGET_EVENT`.

A widget application creates and realizes its widgets, and then calls `XMANAGER`. `XMANAGER` arranges for an event-handling procedure supplied by the application to be called when events for it arrive. The application is shielded from the details of calling `WIDGET_EVENT` and interacting with other widget applications that may be running simultaneously.

For these reasons, widget applications should always use `XMANAGER` in preference to calling `WIDGET_EVENT` directly. The file `xmng_tmpr.pro`, found in the `lib`

subdirectory of the main IDL directory, is a template for writing widget applications that use XMANAGER.

## A Note About Blocking in XMANAGER

Beginning with IDL version 5.0, most versions of IDL's command-processing front-end are able to support an *active command line* while running properly constructed widget applications. What this means is that—provided the widget application is properly configured—the IDL command input line is available for input while a widget application is running and widget events are being processed.

There are currently 5 separate IDL command-processing front-end implementations:

- Apple Macintosh Integrated Development Environment (IDLDE)
- Microsoft Windows IDLDE
- Motif IDLDE (Unix and VMS)
- Unix plain tty
- VMS plain tty

All of these front-ends are able to process widget events except for the VMS plain tty. VMS users can still enjoy an active command line by using the IDLDE interface.

If the command-processing front-end can process widget events (that is, if the front-end is *not* the VMS plain tty), it is still necessary for widget applications to be well-behaved with respect to blocking widget event processing. Since in most cases XMANAGER is used to handle widget event processing, this means that in order for the command line to remain active, all widget applications must be run with the NO\_BLOCK keyword to XMANAGER set. (Note that since NO\_BLOCK is *not* the default, it is quite likely that some application will block.) If a single application runs in blocking mode, the command line will be inaccessible until the blocking application exits. When a blocking application exits, the IDL command line will once again become active.

## JUST\_REG vs. NO\_BLOCK

Although their names imply a similar function, the JUST\_REG and NO\_BLOCK keywords perform very different services. It is important to understand what they do and how they differ.

The JUST\_REG keyword tells XMANAGER that it should simply register a client and then return immediately. The result is that the client becomes known to XMANAGER, and that future calls to XMANAGER will take this client into account. Therefore, JUST\_REG only controls how the registering call to

XMANAGER should behave. The client can still be registered as requiring XMANAGER to block by setting `NO_BLOCK=0`. In this case, *future* calls to XMANAGER will block.

---

**Note**

JUST\_REG is useful in situations where you suspect blocking might occur—if the active command line is not supported and you wish to keep it active before beginning event processing, or if blocking will be requested at a later time. If no blocking will occur or if the blocking behavior is useful, it is not necessary to use JUST\_REG.

---

The `NO_BLOCK` keyword tells XMANAGER that the registered client does not require XMANAGER to block if the command-processing front-end is able to support active command line event processing. XMANAGER remembers this attribute of the client until the client exits, even after the call to XMANAGER that registered the client returns. `NO_BLOCK` is just a “vote” on how XMANAGER should behave—the final decision is made by XMANAGER by considering the `NO_BLOCK` attributes of *all* of its current clients as well as the ability of the command-processing front-end in use to support the active command line.

## Blocking vs. Non-blocking Applications

The issue of blocking in XMANAGER requires some explanation. IDL widget events are not processed until the `WIDGET_EVENT` function is called to handle them. Otherwise, they are queued by IDL indefinitely. Knowing how and when to call `WIDGET_EVENT` is the primary service provided by XMANAGER.

There are two ways blocking is typically handled:

1. The first call to XMANAGER processes events by calling `WIDGET_EVENT` as necessary until no managed widgets remain on the screen. This is referred to as “blocking” because XMANAGER does not return to the caller until it is done, and the IDL command line is not available.
2. XMANAGER does not block, and instead, the part of IDL that reads command input also watches for widget events and calls `WIDGET_EVENT` as necessary while also reading command input. This is referred to as “non-blocking” or “active command line” mode.

XMANAGER will block unless all of the following conditions are met:

- The command-processing front-end is able to process widget events (that is, the front-end is not the VMS plain tty).

- All registered widget applications have the `NO_BLOCK` keyword to `XMANAGER` set.
- No modal dialogs are displayed. (Modal dialogs always block until dismissed.)

In general, we suggest that new widget applications be written with `XMANAGER` blocking disabled (that is, with the `NO_BLOCK` keyword set). Since a widget application that does block event processing for itself will block event processing for all other widget applications (and the IDL command line) as well, we suggest that older widget applications be upgraded to take advantage of the new, non-blocking behavior by adding the `NO_BLOCK` keyword to most calls to `XMANAGER`.

## Features Reserved to `XMANAGER`

Because `XMANAGER` buffers you from direct handling of widget events, it requires that you not explicitly specify certain event handling features for the top-level base of your widget application. To be able to work with `XMANAGER`, widget applications should avoid the following pitfalls. If you ignore the suggestions below, your changes will conflict with those made by `XMANAGER`:

- Do not specify an event-handling function or procedure on the top-level base of a widget application using the `EVENT_FUNC` or `EVENT_PRO` keywords to the widget creation functions or `WIDGET_CONTROL`. Instead, provide the name of the event handler routine to `XMANAGER` via the `EVENT_HANDLER` keyword.
- Do not specify a death notification procedure on the top-level base of a widget application using the `KILL_NOTIFY` keyword to the widget creation functions or `WIDGET_CONTROL`. Instead, provide the name of your “cleanup” routine to `XMANAGER` via the `CLEANUP` keyword.

For a detailed discussion of `XMANAGER`, see [XMANAGER](#) in the *IDL Reference Guide*.

## The `XREGISTERED` Function

The `XMANAGER` procedure does not restrict applications to only a single running copy. Indeed, it is desirable for most applications to allow multiple simultaneous instances to run. However, there are some applications that should only allow a single instance at a time, either because it makes logical sense or because a weakness in the implementation requires it. An obvious example of this is an application that uses a `COMMON` block to maintain its current state (see “[Managing Widget Application State](#)” on page 592).

The `XREGISTERED` function can be used in such applications to ensure that only a single copy can run at a time. Place the following statement at the start of the routine:

IF XREGISTERED('routine\_name') THEN RETURN

where *routine\_name* is the name of the widget application.

See [XREGISTERED](#) in the *IDL Reference Guide* for further information.



## Widget Example 2

The following example demonstrates how user values can be used to simplify event processing and to pass variables between routines. It creates a base widget with three buttons and a text field that reports which button was pressed.

Enter the two procedures listed below — either in a text file named `widget2.pro`, or directly into IDL using the `.RUN` command. Enter `widget2` at the IDL prompt to run the program.

```

PRO widget2_event, ev
WIDGET_CONTROL, ev.top, GET_UVALUE=textwid
WIDGET_CONTROL, ev.id, GET_UVALUE=uval
CASE uval OF
  'ONE' : WIDGET_CONTROL, textwid, SET_VALUE='Button One Pressed'
  'TWO' : WIDGET_CONTROL, textwid, SET_VALUE='Button Two Pressed'
  'DONE': WIDGET_CONTROL, ev.top, /DESTROY
ENDCASE
END

PRO widget2
base = WIDGET_BASE(/COLUMN)
button1 = WIDGET_BUTTON(base, VALUE='One', UVALUE='ONE')
button2 = WIDGET_BUTTON(base, VALUE='Two', UVALUE='TWO')
text = WIDGET_TEXT(base, XSIZE=20)
button3 = WIDGET_BUTTON(base, value='Done', UVALUE='DONE')
WIDGET_CONTROL, base, SET_UVALUE=text
WIDGET_CONTROL, base, /REALIZE
XMANAGER, 'Widget2', base
END

```

Alternately, you can run the program from the IDL distribution by entering:

```
widget2
```

at the IDL command prompt. See [“Running the Example Code”](#) on page 543 if IDL does not run the program as expected.

Once again, let’s examine the creation routine, `widget2`, first. We first create a top-level base, this time specifying the `COLUMN` keyword to ensure that the widgets contained in the base are stacked vertically. We create two buttons with values “One” and “Two,” and user values “ONE” and “TWO.” Remember that the value of a button widget is also the button’s label. We create a text widget, and specify its width to be 20 characters using the `XSIZE` keyword. The last button is the “Done” button, with a the user value “DONE.”

Next follow two calls to the `WIDGET_CONTROL` procedure. The first sets the user value of the top-level base equal to the widget ID of our text widget. This will allow us easy access to the text widget from our event handling routine. The second realizes the top-level base and all its child widgets. Finally, we invoke the `XMANAGER` to manage the widget application.

The `widget2_event` routine is slightly more complicated than its predecessor. We begin by using `WIDGET_CONTROL` to retrieve the widget ID of the our text widget from the user value of the top-level base. We can do this because we know that the widget ID of our top-level base is contained in the `TOP` field of the widget event structure — thus, `ev.top` contains the widget ID of the base widget. We use the `GET_UVALUE` keyword to store the widget ID of the text widget in the variable `textwid`.

Next, we use `WIDGET_CONTROL` and the `GET_UVALUE` keyword to retrieve the user value of the widget that generated the event. Again, we can do this because we know that the widget ID of the widget that generated the event is stored in the `ID` field of the event structure. We then use a `CASE` statement to compare the user value of the widget, now stored in the variable `uval`, with the list of possible user values (which we know, have set them explicitly in the creation routine) to determine which button was pressed and act accordingly.

In the `CASE` statement, we check to see if `uval` is the user value associated with either button one or button two. If it is, we use `WIDGET_CONTROL` and the `SET_VALUE` keyword to alter the value of the text widget, whose ID we stored in the variable `textwid`. If `uval` is 'DONE', we recognize that the user has clicked on the “Done” button and use `WIDGET_CONTROL` to destroy the widget hierarchy.

# Using Draw Widgets

Draw widgets are graphics windows that appear as part of a widget hierarchy rather than appearing as an independent window. Like other graphics windows, draw widgets can be created to use either Direct or Object graphics. (See [Chapter 10, “Graphics”](#) in the *Using IDL* manual for a discussion of IDL’s two graphics modes.) Draw widgets allow designers of IDL graphical user interfaces to take advantage of the full power of IDL graphics in their displays.

## Using Direct Graphics in Draw Widgets

Standard Direct graphics windows are created using the `WINDOW` procedure, while Direct graphics draw widgets are created using the `WIDGET_DRAW` function with the `GRAPHICS_LEVEL` keyword set equal to one. Draw widgets use Direct graphics by default. Once created, Direct graphics windows and draw widgets are used in the same way.

All IDL Direct graphics windows are referred to by a window number. Unlike windows created by the `WINDOW` procedure, the window number of a Direct graphics draw widget cannot be assigned by the user. In addition, the window number of a draw widget is not assigned until the draw widget is actually realized, and thus cannot be returned by `WIDGET_DRAW` when the widget is created. Instead, you must use the `WIDGET_CONTROL` procedure to retrieve the window number, which is stored in *value* of the draw widget, *after* the widget has been realized.

Unlike normal graphics windows, creating a draw widget does not cause the current graphics window to change to the new widget. You must use the `WSET` procedure to explicitly make the draw widget the current graphics window. The following IDL statements demonstrate the required steps:

```
;Create a base widget.
base = WIDGET_BASE()

;Attach a 256 x 256 draw widget.
draw = WIDGET_DRAW(base, XSIZE = 256, YSIZE = 256)
;Realize the widgets.
WIDGET_CONTROL, /REALIZE, base

;Obtain the window index.
WIDGET_CONTROL, draw, GET_VALUE = index

;Set the new widget to be the current graphics window
WSET, index
```

If you attempt to get the value of a draw widget before the widget has been realized, IDL returns the value -1, which is not a valid index.

## Using Object Graphics in Draw Widgets

Standard Object graphics windows are IDLgrWindow objects, whereas Object graphics draw widgets are created using the `WIDGET_DRAW` function with the `GRAPHICS_LEVEL` keyword set equal to two. Once created, Object graphics windows and draw widgets are used in the same way.

All IDL Object graphics windows are referred to by an object reference. Since you do not explicitly create the IDLgrWindow object used in a draw widget, you must retrieve the object reference by using the `WIDGET_CONTROL` procedure to get the *value* of the draw widget. As with Direct graphics draw widgets, the window object is not created—and thus the object reference cannot be retrieved—until after the draw widget is realized.

## Scrolling Draw Widgets

Another difference between a draw widget and either a graphics window created with the `WINDOW` procedure or an IDLgrWindow object is that draw widgets can include scroll bars. Setting the `APP_SCROLL` keyword or the `SCROLL` keyword to the `WIDGET_DRAW` function causes scrollbars to be attached to the drawing widget, which allows the user to view images or graphics larger than the visible area. Use the `APP_SCROLL` keyword when displaying images, or anything drawn in device units or pixels. Use the `SCROLL` keyword when a draw widget is going to display graphics drawn in data units (e.g., `PLOT`, `CONTOUR`, `SURFACE`).

The IDL `SLIDE_IMAGE` routine is an example of a widget application that uses both regular and scrolling draw widgets. See [WIDGET\\_DRAW](#) in the *IDL Reference Guide* for details, or inspect the file `slide_image.pro` in the `lib` subdirectory of your main IDL directory for an example.

# Creating Menus

Menus allow a user to select one or more options from a list of options. IDL widgets allow you to build a number of different types of menus for your widget application.

## Button Groups

One approach to menu creation is to build an array of buttons. With a button menu, all options are visible to the user all the time. To create a button menu, do the following:

1. Call the `WIDGET_BASE` function to create a base to hold the buttons. Use the `COLUMN` and `ROW` keywords to determine the layout of the buttons.
2. Call the `WIDGET_BUTTON` function once for each button to be added to the base created in the previous step.

Because menus of buttons are common, IDL provides a compound widget named `CW_BGROU`P to create them. Using `CW_BGROU`P rather than a series of calls to `WIDGET_BUTTON` simplifies creation of a menu of buttons and also simplifies event handling by providing a single event structure for the group of buttons. For example, the following IDL statements create a button menu with five choices:

```
values = ['One', 'Two', 'Three', 'Four', 'Five']
base = WIDGET_BASE()
bgroup = CW_BGROU(base, VALUE=values, /COLUMN)
WIDGET_CONTROL, base, /REALIZE
```

In this example, one call to `CW_BGROU`P replaces five calls to `WIDGET_BUTTON`.

## Exclusive or Nonexclusive Buttons

Buttons in button groups normally act as independent entities, returning a selection event (a one in the select field of the event structure) or similar value when pressed. Groups of buttons can also be made to act in concert, as either exclusive or non-exclusive groups. In contrast to normal button groups, both exclusive and non-exclusive groups display which buttons have been selected.

*Exclusive* button groups allow only one button to be selected at a given time. Clicking on an unselected button deselects any previously-selected buttons. *Non-exclusive* button groups allow any number of buttons to be selected at the same time. Clicking on the same button repeatedly selects and deselects that button.

The following code creates three button groups. The first group is a “normal” button group as created in the previous example. The next is an exclusive group, and the third is a non-exclusive group.

```
values = ['One', 'Two', 'Three', 'Four', 'Five']
base = WIDGET_BASE(/ROW)
bgroup1 = CW_BGROUП(base, VALUE=values, /COLUMN, $
    LABEL_TOP='Normal', /FRAME)
bgroup2 = CW_BGROUП(base, VALUE=values, /COLUMN, /EXCLUSIVE, $
    LABEL_TOP='Exclusive', /FRAME)
bgroup3 = CW_BGROUП(base, VALUE=values, /COLUMN, /NONEXCLUSIVE, $
    LABEL_TOP='Nonexclusive', /FRAME)
WIDGET_CONTROL, base, /REALIZE
```

The widget created by this code is shown in the following figure:



Figure 18-2: Normal Menu (left), Exclusive Menu (center) and Non-exclusive Menu (right)

## Lists

A second approach to menu creation is to provide the user with a list of options in the form of a scrolling or drop-down list. A scrolling list is always displayed, although it may not show all items in the list at all times. A drop-down list shows only the selected item until the user clicks on the list, at which time it displays the entire list. Both lists allow only a single selection at a time.

The following example code uses the `WIDGET_LIST` and `WIDGET_DROPLIST` functions to create two menus of five items each. While both lists contain five items, the scrolling list displays only three at a time, because we specify this with the `YSIZE` keyword.

```

values = ['One', 'Two', 'Three', 'Four', 'Five']
base = WIDGET_BASE(/ROW)
list = WIDGET_LIST(base, VALUE=values, YSIZE=3)
drop = WIDGET_DROPLIST(base, VALUE=values)
WIDGET_CONTROL, base, /REALIZE

```

The widget created by this code is shown in the following figure:

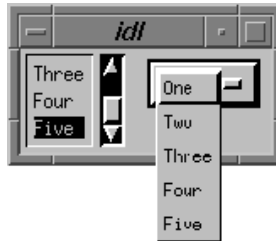


Figure 18-3: A scrolling list and a drop-down list.

## Pulldown Menus

A third approach to menu creation involves menus that appear as a single button until the user selects the menu, at which time the menu pops up to display the list of possible selections. Buttons in such a pulldown menu can activate other pulldown menus to any desired depth. The method for creating a pulldown menu is as follows:

1. The topmost element of any pulldown menu is a button, created with the MENU keyword to the WIDGET\_BUTTON function.
2. The top-level button has one or more child widget buttons attached. (That is, one or more buttons specify the first button's widget ID as their "parent.") Each button can either be used as is, in which case pressing it causes an event to be generated, or it can be created with the MENU keyword and have further child widget buttons attached to it. If it has child widgets, pushing it causes a pulldown menu containing the child buttons to pop into view.
3. Menu buttons can be the parent of other buttons to any desired depth.

Because pulldown menus are common, IDL provides a compound widget named CW\_PDMENU to create them. Using CW\_PDMENU rather than a series of calls to WIDGET\_BUTTON simplifies creation of a pulldown menu in the same way the CW\_BGROUP simplifies the creation of button menus.

The following example uses `CW_PDMENU` to create a pulldown menu. First, we create an array of anonymous structures to contain the menu descriptions.

```
desc = REPLICATE({ flags:0, name:'' }, 6)
```

The `desc` array contains six copies of the empty structure. Each structure has two fields: `flags` and `name`. Next, we populate these fields with values:

```
desc.flags = [ 1, 0, 1, 0, 2, 2 ]
desc.name = [ 'Operations', 'Predefined', 'Interpolate', $
             'Linear', 'Spline', 'Quit' ]
```

The value of the `flags` field specifies the role of each button. In this example, the first and third buttons start a new sub-menu (values are 1), the second and fourth buttons are plain buttons with no other role (values are 0), and the last two buttons end the current sub-menu and return to the previous level (values are 2). The value of the `name` field is the value (or label) of the button at each level.

```
base = WIDGET_BASE()
menu = CW_PDMENU(base, desc)
WIDGET_CONTROL, base, /REALIZE
```

The format of the menu description used by `CW_PDMENU` in the above example requires some explanation. `CW_PDMENU` views a menu as consisting of a series of buttons, each of which can optionally lead to a sub-menu. The description of each button consists of a structure supplying its name and a flag field that tells what kind of button it is (starts a new sub-menu, ends the current sub-menu, or a plain button within the current sub-menu). The description of the complete menu consists of an array of such structures corresponding to the flattened menu. Compare the description used in the code above with the result shown in the following figure.

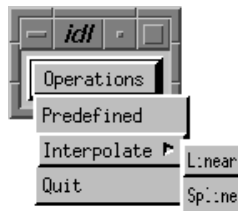


Figure 18-4: Pulldown menu created with `CW_PDMENU`.



## Menus on Top-Level Bases

A final approach to providing menus in your widget application is to attach the menus directly to the top-level base widget. Menus attached to a top-level base widget are created just like pulldown menus created from button widgets, but they do not appear as buttons. Menus created in this way are children of a special sub-base of the top-level base, created by specifying the MBAR keyword when the top-level base is created.

For example, the following code creates a top-level base widget and attaches a menu titled MENU1 to it. MENU1 contains the choices ONE, TWO, and THREE.

```
base = WIDGET_BASE(MBAR=bar)
menu1 = WIDGET_BUTTON(bar, VALUE='MENU1', /MENU)
button1 = WIDGET_BUTTON(menu1, VALUE='ONE')
button2 = WIDGET_BUTTON(menu1, VALUE='TWO')
button3 = WIDGET_BUTTON(menu1, VALUE='THREE')
draw = WIDGET_DRAW(base, XSIZE=100, YSIZE=100)
WIDGET_CONTROL, base, /REALIZE
```

The resulting widget is shown in the following figure:



Figure 18-5: Menu attached to a top-level base.

# Controlling Widgets

The `WIDGET_CONTROL` procedure allows you to realize, manage, and destroy widget hierarchies. It is often used to change the default behavior or appearance of previously-realized widgets.

Some keywords to `WIDGET_CONTROL` affect only certain types of widgets, others affect any type of widget, and some affect the widget system in general without being tied to a single widget ID or widget type. See [WIDGET\\_CONTROL](#) in the *IDL Reference Guide* for complete details. We discuss here only a few of the more common uses of this procedure.

## Realizing Widget Hierarchies

As we have seen in the above examples, widgets must be *realized* before they appear on screen. In most cases, you will want to realize your entire widget hierarchy at the same time. Do this with the statement

```
WIDGET_CONTROL, base, /REALIZE
```

where `base` is the widget ID of the top-level base widget for your widget hierarchy.

## Killing Widget Hierarchies

The standard way to kill a widget hierarchy is with the statement

```
WIDGET_CONTROL, base, /DESTROY
```

where `base` is the widget ID of the top-level base widget of the hierarchy to be killed. Usually, IDL programs that use widgets issue this statement in their event-handling routine in response to the application “Done” button.

In addition, some window managers place a pulldown menu on the frame of the top-level base widget that allows the user to kill the entire hierarchy. Using the window manager to kill a widget hierarchy is equivalent to using the `WIDGET_CONTROL` procedure.

When designing widget applications, you should always include a “Done” button in the application itself, because some window managers do not provide the user with a kill option from the outer frame.

## Retrieving or Changing Widget Values

As we discussed previously, you can use `WIDGET_CONTROL` to retrieve or change widget values using the `GET_VALUE` and `SET_VALUE` keywords. Similarly, you

can retrieve or change widget user values with the `GET_UVALUE` and `SET_UVALUE` keywords.

For example, you could use the following command in an event handling procedure to save the user value of the widget that generates an event into an IDL variable named `uval`:

```
WIDGET_CONTROL, event.id, GET_UVALUE=uval
```

Similarly, you could use the following commands to retrieve the value of a draw widget named `drawwid` and make that draw widget the current graphics window:

```
WIDGET_CONTROL, drawwid, GET_VALUE=draw  
WSET, draw
```

## Sensitizing Widgets

When a widget is sensitive, it has normal appearance and can receive user input. When a widget is insensitive, it ignores any input directed at it. Use sensitivity to control when a user is allowed to manipulate a widget. Note that while most widgets change their appearance when they become insensitive, some simply stop generating events.

Set the `SENSITIVE` keyword equal to zero to desensitize a widget, or to a nonzero value to make it sensitive. For example, you might wish to make a group of buttons named `bgroup` insensitive after some user input. You would use the following command:

```
WIDGET_CONTROL, bgroup, SENSITIVE=0
```

## Indicating Time-Consuming Operations

In an event driven environment, it is important that the interface be highly responsive to the user's manipulations. This means that widget event handlers should be written to execute quickly and return. However, sometimes the event handler has no option but to perform an operation that is slow. In such a case, it is a good idea to give the user feedback that the system is busy. This is easily done using the `HOURLASS` keyword just before the expensive operation is started:

```
WIDGET_CONTROL, /HOURLASS
```

This command causes IDL to turn on an hourglass-shaped cursor for all IDL widgets and graphics windows. The hourglass remains active until the next event is processed, at which point the previous cursor is automatically restored.

## Using Timer Events

In addition to the normal widget events discussed previously, IDL allows the user to make *timer event* requests by using the `TIMER` keyword. Such events are useful in many applications that are time dependent, such as animation. The syntax for making such a request is:

```
WIDGET_CONTROL, Widget_Id, TIMER=interval_in_seconds
```

*Widget\_Id* can be the ID of any type of widget. When such a request is made, IDL generates a timer request after the requested time interval has passed. Timer events consist of a structure with only the standard three fields — no additional information is provided.

It is up to the programmer to differentiate between a normal event and a timer event for a given widget. The usual way to solve this problem is to make timer requests for widgets that do not otherwise generate events, such as base or label widgets.

Each timer request causes a single event to be generated. To generate a steady stream of timer events, you must make a new timer request in the event handler routine each time a timer event is delivered.

## Widget Example 3

The following example program creates a small widget application consisting of a draw widget and a droplist menu. One of three plots is displayed in the draw widget depending on the selection made from the droplist. To add to the excitement, we will use timer events to change the color table used in the draw window every three seconds.

Enter the two procedures listed below — either in a text file named `widget3.pro`, or directly into IDL using the `.RUN` command. Enter `widget3` at the IDL prompt to run the program.

```
PRO widget3_event, ev
```

We need to save the value of the seed variable for the random number generator between calls to the event-handling routine. We do this using a `COMMON` block.

```
COMMON wid3, seed
```

Retrieve the widget ID of the draw widget and make it the current IDL graphics window:

```
WIDGET_CONTROL, ev.top, GET_UVALUE=drawID
WSET, drawID
```

Check the type of event structure returned. If it is a timer event, change the color table index to a random number between 0 and 40. (See [“Event Processing And Callbacks”](#) on page 589 for more on identifying widget types from returned event structures.)

```
IF (TAG_NAMES(ev, /STRUCTURE_NAME) EQ 'WIDGET_TIMER') $
  THEN BEGIN
    ;Pick a random number.
    table = FIX(RANDOMU(seed)*41)
    ;Load the color table.
    LOADCT, table
    ;Reset the timer.
    WIDGET_CONTROL, ev.id, TIMER=3.0
  ENENDIF
```

If the event is a droplist event, change the type of plot displayed in the draw widget. Note the use of the `index` field of events returned from the droplist widget to determine the value selected.

```
IF (TAG_NAMES(ev, /STRUCTURE_NAME) EQ 'WIDGET_DROPLIST') $
  THEN BEGIN
    CASE ev.index OF
```

```

0: PLOT, DIST(150)
1: SURFACE, DIST(150)
2: SHADE_SURF, DIST(150)
3: WIDGET_CONTROL, ev.top, /DESTROY
ENDCASE
ENDIF
END

```

Create a base widget containing a draw widget and a droplist menu.

```

PRO widget3

select = ['Plot', 'Surface', 'Shaded Surface', 'Done']
base = WIDGET_BASE(/COLUMN)
draw = WIDGET_DRAW(base, XSIZE=150, YSIZE=150)
dlist = WIDGET_DROPLIST(base, VALUE=select)

```

Realize the widget hierarchy, then retrieve the value of the draw widget and store it in the user value of the base widget. (Note that we are using Direct graphics for the draw widget, so the value is an IDL graphics window ID.) Finally, set the timer value of the draw widget.

```

WIDGET_CONTROL, base, /REALIZE
WIDGET_CONTROL, draw, GET_VALUE=drawID
WIDGET_CONTROL, base, SET_UVALUE=drawID
WIDGET_CONTROL, draw, TIMER=0.0

```

Set the droplist to display “Shaded Surface” and place a shaded surface in the draw widget:

```

WIDGET_CONTROL, dlist, SET_DROPLIST_SELECT=2
WSET, drawID
SHADE_SURF, DIST(150)
;Register the widget with the XMANAGER.
XMANAGER, 'widget3', base
END

```

Alternately, you can run the program from the IDL distribution by entering:

```

widget3

```

at the IDL command prompt. See [“Running the Example Code”](#) on page 543 if IDL does not run the program as expected.

This example is intentionally silly. The intent is to demonstrate the use of draw widgets, menus, and timer events with a minimum of other issues to complicate things. However, it is easy to imagine applications wherein a graphics window containing a plot or some other information is updated periodically by a timer. The method used here can be easily applied to more realistic situations.

# Widget Sizing

This section explains how IDL widgets size themselves, widget geometry concepts, and how to explicitly size and position widgets.

## Widget Geometry Terms and Concepts

Widget geometry, or the size and layout of widgets, is determined by many interrelated factors. In the following discussion, the following terms are used:

- *Geometry*: The size and position of a widget.
- *Natural Size*: The natural, or implicit, size of a widget is the size a widget has if no external constraints are placed on it. For example, a label widget has a natural size that is determined by the size of the text it is displaying and space for margins. These values are influenced by such things as the size of the font being displayed and characteristics of the low-level (i.e., operating-system level) widget or control used to implement the IDL widget.
- *Explicit Size*: The explicit, or user-specified, size of a widget is the size set when an IDL programmer specifies one of the size keywords to an IDL widget creation function or `WIDGET_CONTROL`.

## How Widget Geometry is Determined

IDL uses the following rules to determine the geometry of a widget:

- The explicit size of a widget, if one is specified, takes precedence over the natural size. That is, the user-specified size is used if available.
- If an explicit size is not specified, the natural size of the widget—at the time the widget is realized—is used. Once realized, the size of a widget *does not automatically change* when the value of the widget changes, unless the widget’s dynamic resize property has been set. Dynamic resizing is discussed in more detail below. Note that any realized widget can be made to change its size by calling `WIDGET_CONTROL` with any of the sizing keywords.
- Children of a “bulletin board” base (i.e., a base that was created without setting the `COLUMN` or `ROW` keywords) have an offset of (0,0) unless an offset is explicitly specified via the `XOFFSET` or `YOFFSET` keywords.
- The offset keywords to widgets that are children of `ROW` or `COLUMN` bases are ignored, and IDL calculates the offsets to lay the children out in a grid.

This calculation can be influenced by setting any of the `ALIGN` or `BASE_ALIGN` keywords when the widgets are created.

## Dynamic Resizing

Realized widgets, by default, do not automatically resize themselves when their values change. This is true whether the widget was created with an explicit size or the widget was allowed to size itself naturally. This behavior makes it easy to create widget layouts that don't change size too frequently or "flicker" due to small changes in a widget's natural size.

This default behavior can be changed for label, button, and droplist widgets. Set the `DYNAMIC_RESIZE` keyword to `WIDGET_LABEL`, `WIDGET_BUTTON`, or `WIDGET_DROPLIST` to make a widget that automatically resizes itself when its value changes. Note that the `XSIZE` and `YSIZE` keywords should not be used with `DYNAMIC_RESIZE`. Setting explicit sizing values overrides the dynamic resize property and creates a widget that *will not* resize itself.

## Explicitly Specifying the Size and Location of Widgets

The `XSIZE` (and `SCR_XSIZE`), `YSIZE` (and `SCR_YSIZE`), `XOFFSET`, and `YOFFSET` keywords, when used with a standard base widget parent (a base created without the `COLUMN` or `ROW` keywords—also called a "bulletin board" base), allow you to specify exactly how the child widgets should be positioned. Sometimes this is a very useful option. However, in general, it is best to avoid this style of programming. Although these keywords are usually honored, they are merely hints to the widget toolkit and might be ignored.

Explicitly specifying the size and offset makes a program inflexible and unable to run gracefully on various platforms. Often, a layout of this type will look good on one platform, but variations in screen size and how the toolkit works will cause widgets to overlap and not look good on another platform. The best way to handle this situation is to use nested row and column bases to hold the widgets and let the widgets arrange themselves. Such bases are created using the `COLUMN` and `ROW` keywords to the `WIDGET_BASE` function.

## Sizing Keywords

When explicitly setting the size of a widget, IDL allows you to control three aspects of the size:

- The *virtual size* is the size of the *potentially* viewable area of the widget. The virtual size may be larger than the actual viewable area on your screen. The



virtual size of a widget is determined by either the widget's value, or the `XSIZE` and `YSIZE` keywords to the widget creation routine.

- The *viewport size* is the size of the viewable area on your screen. If the viewport size is smaller than the virtual size, scroll bars may be present to allow you to view different sections of the viewable area. When creating widgets for which scroll bars are appropriate, you can add scroll bars by setting the `SCROLL` keyword to the widget creation routine. You can explicitly set the size of the viewport area using the `X_SCROLL_SIZE` and `Y_SCROLL_SIZE` keywords when creating base, draw, and table widgets.

---

**Note**

With draw widgets, you can set the `APP_SCROLL` or the `SCROLL` keyword. Use the `APP_SCROLL` keyword when displaying images, or anything drawn in device units or pixels. Use the `SCROLL` keyword when a draw widget is going to display graphics drawn in data units (e.g., `PLOT`, `CONTOUR`, `SURFACE`).

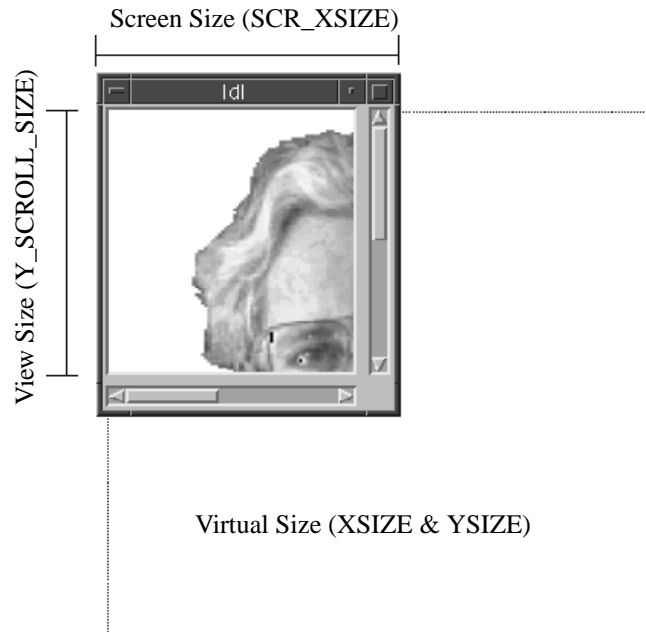
---

- The *screen size* is the size of the widget on your screen. You can explicitly specify a screen size using the `SCR_XSIZE` and `SCR_YSIZE` keywords to the widget creation routine. Explicitly-set viewport sizes (set with `X_SCROLL_SIZE` or `Y_SCROLL_SIZE`) are ignored if you specify the screen size.

The following code shows an example of the `WIDGET_DRAW` command:

```
draw = WIDGET_DRAW(base, XSIZE=384, YSIZE=384, $
    X_SCROLL_SIZE=192, Y_SCROLL_SIZE = 192, SCR_XSIZE=200)
```

This results in the following:



*Figure 18-6: Visual description of widget sizes.*

In this case, the `XSIZE` and `YSIZE` keywords set the virtual size to 384 x 384 pixels. The `X_SCROLL_SIZE` and `Y_SCROLL_SIZE` keywords set the viewport size to 192 x 192 pixels. Finally, the `SCR_XSIZE` keyword overrides the `X_SCROLL_SIZE` keyword and forces the screen size of the widget (in the X-dimension) to 200 pixels, including the scroll bar.

### Controlling Widget Size after Creation

A number of keywords to the `WIDGET_CONTROL` procedure allow you to change the size of a widget after it has been created. (You will find a list of the keywords to `WIDGET_CONTROL` that apply to each type of widget at the end of the widget creation routine documentation.) Note that keywords to `WIDGET_CONTROL` may not control the same parameters as their counterparts associated with widget creation routines. For example, while the `XSIZE` and `YSIZE` keywords to `WIDGET_DRAW` control the virtual size of the draw widget, the `XSIZE` and `YSIZE` keywords to `WIDGET_CONTROL` (when called with the widget ID of a draw widget) control the viewport size of the draw widget.

## Units of Measurement

You can specify the unit of measurement used for most widget sizing operations. When using a widget creation routine, or when using `WIDGET_CONTROL` or `WIDGET_INFO`, set the `UNITS` keyword 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.

---

### Note

The `UNITS` keyword does not affect all sizing operations. Specifically, the value of `UNITS` is ignored when setting the `XSIZE` or `YSIZE` keywords to `WIDGET_LIST`, `WIDGET_TABLE`, or `WIDGET_TEXT`.

---

## Finding the Size of the Screen

When creating the top-level base for an application, sometimes it is useful to know the size of the screen. This information is available via the `GET_SCREEN_SIZE` function. `GET_SCREEN_SIZE` returns a two-element integer array specifying the size of the screen, in pixels. See `GET_SCREEN_SIZE` in the *IDL Reference Guide* for details.

## Preventing Layout Flicker

After a widget hierarchy has been realized, adding or destroying widgets in that hierarchy causes IDL to recalculate and set new geometries for every widget in the hierarchy. When a number of widgets are added or destroyed, these calculations occur between each change to the hierarchy, resulting in unpleasant screen “flashing” as the user sees a brief display of each intermediate widget configuration. This behavior can be eliminated by using the `UPDATE` keyword to `WIDGET_CONTROL`.

The top-level base of every widget hierarchy has an `UPDATE` attribute that determines whether or not changes to the hierarchy are displayed on screen. Setting `UPDATE` to 0 turns off immediate updates and allows you to make a large number of changes to a widget hierarchy without updating the screen after each change. After all of your changes have been made, setting `UPDATE` to 1 causes the final widget configuration to be displayed on screen.

For example, consider the following main-level program that realizes an unmapped base, then adds 200 button widgets to the previously-realized base:

```
time = SYSTIME(1)
```

```

b = WIDGET_BASE(/COLUMN, XSIZE=400, YSIZE=400, MAP=0)
WIDGET_CONTROL, b, /REALIZE
FOR i = 0, 200 DO button = WIDGET_BUTTON(b, VALUE=STRING(i))
WIDGET_CONTROL, b, /MAP
PRINT, 'time used: ', SYSTIME(1) - time
END

```

This program takes over 50 seconds to run on an HP 9000/720 workstation. If the base had been mapped, the user would see the base “flashing” as each button was added to the base. Altering the example to use the UPDATE keyword reduces the execution time to 0.7 seconds:

```

time = SYSTIME(1)
b = WIDGET_BASE(/COLUMN, XSIZE=400, YSIZE=400, MAP=0)
WIDGET_CONTROL, b, /REALIZE, UPDATE=0
FOR i = 0, 200 DO button = WIDGET_BUTTON(b, VALUE=STRING(i))
WIDGET_CONTROL, b, /MAP, /UPDATE
PRINT, 'time used: ', SYSTIME(1) - time
END

```

---

### Note

Do not attempt to resize a widget on the Windows platform while UPDATE is turned off. Doing so may prevent IDL from updating the screen properly.

---

# Event Processing And Callbacks

Previously we mentioned that when IDL receives an event, it is queued until a call to `WIDGET_EVENT` is made, when the event is dequeued and returned. That is a simplified description of what actually happens.

All events for a given widget are processed in the order that they are generated. The event processing performed by `WIDGET_EVENT` consists of the following steps, applied iteratively:

- `WIDGET_EVENT` waits for an event from one of the specified widgets to arrive.
- Starting with the widget that the event belongs to, move up the widget hierarchy looking for a widget that has an event-handling procedure or function associated with it. Such routines are associated with a widget via the `EVENT_PRO` and `EVENT_FUNC` keywords to the widget creation functions or the `WIDGET_CONTROL` procedure.
- If an event-handling *procedure* is found, it is called with the event as its argument. The `HANDLER` field of the event is set to the widget ID of the widget associated with the handling procedure. When the procedure returns, `WIDGET_EVENT` returns to the first step above and starts searching for events. Hence, event-handling procedures are said to “swallow” events.
- If an event-handling *function* is found, it is called with the event as its argument. The `HANDLER` field of the event is set to the widget ID of the widget associated with the handling function.

When the function returns, its value is examined. If the value is not a structure, it is discarded and `WIDGET_EVENT` returns to the first step. This behavior allows event-handling functions to selectively act like event-handling procedures and “swallow” events.

If the returned value is a structure, it is checked to ensure that it has the standard first 3 fields: `ID`, `TOP`, and `HANDLER`. If not, an error is issued. Otherwise, the returned value replaces the event found in the first step and `WIDGET_EVENT` continues moving up the widget tree looking for another event handler routine, as described in step 2, above.

Hence, event functions are said to “rewrite” events. This ability to rewrite events is the basis of *compound widgets* which combine several widgets to give the appearance of a single, more complicated widget. Compound widgets are an important widget programming concept. For more information, see [“Compound Widgets”](#) on page 594.

- If an event reaches the top of a widget hierarchy without being swallowed by an event handler, it is returned as the value of `WIDGET_EVENT`.
- If `WIDGET_EVENT` was called without an argument, and there are no widgets left on the screen that are being managed (as set via the `MANAGED` keyword to the `WIDGET_CONTROL` procedure) and could generate events, `WIDGET_EVENT` ends the search and returns an empty event (i.e., a standard widget event structure with the top three fields set to zero).

## Identifying Widget Type from an Event

Given a widget event structure, often you need to know what type of widget generated it without having to match the widget ID in the event structure to all the current widgets. This information is available by specifying the `STRUCTURE_NAME` keyword to the `TAG_NAMES` function:

```
PRINT, 'Event structure type: ', $
      TAG_NAMES (EVENT, /STRUCTURE_NAME)
```

This works because each widget type generates a different event structure. The event structure generated by a given widget type is documented in the description of the widget creation function for that type.

When using this technique, be aware that although all the basic widgets use named structures for their events, many compound widgets return anonymous structures. This technique does not work well in that case because anonymous structures lack a recognizable name.

### Note

---

Always check for a distinct type of widget event. Research Systems will continue to add new widgets with new event structures, so it is important not to make assumptions about the contents of a random widget event structure. The structure of existing widget events will remain stable, so checking for a particular type of widget event will always work.

---

## Keyboard Focus Events

Base, table, and text widgets can be set to generate *keyboard focus events*. Generating and examining keyboard focus events allows you to determine when a given widget has either *gained* or *lost* the keyboard focus—that is, when it is brought to the foreground or when it is covered by another window.

Set the `KBRD_FOCUS_EVENTS` keyword to `WIDGET_BASE`, `WIDGET_TABLE`, or `WIDGET_TEXT` to generate keyboard focus events. You can then use your event-handling procedure to cache the widget ID of the last widget (with keyboard focus events enabled) to have the keyboard focus. One situation where this is useful is when you have an application menu (created with the `MBAR` or `APP_MBAR` keyword to `WIDGET_BASE`) and you wish to perform an action in a text widget based on the menu item selected. Although the event generated by the user's menu selection has the *menu's* base as its top-level widget ID, if you generate and track keyboard focus events for the text widget, you can “remember” which widget the action triggered by the menu selection should affect. Note that in this example, keyboard focus events are *not* generated for the menubar's base.

## Interrupting the Event Loop

Beginning with IDL version 5, IDL has the ability to process commands from the IDL command line while simultaneously processing widget events. This means that the IDL command will remain active even when widget applications are running.

It is possible to interrupt the event function by sending the interrupt character (Control-C or Command-C). However, you may find that even after sending the interrupt, IDL does not immediately interrupt the event loop. IDL will interrupt the process that is “on top”—that is, if several applications are running at once, the interrupt will be handled by the first application to receive it.

If your widget application is the only active application, and sending the interrupt does not cause it to break, move the mouse cursor across (or click on) one of the widgets.

This works because when IDL is in the event function, it only checks for the interrupt between event notifications from the window system. Such events do not necessarily translate one-to-one into IDL widget events because the window system typically generates a large number of events related to the window system's operation that IDL quietly handles. Moving the mouse cursor across the widgets typically generates some of these events which gives IDL a chance to notice the interrupt and act on it.

---

### Note

Do not interrupt the event loop by placing a `STOP` or `EXIT` command in the event-handler or in a callback routine. The presence of either command will cause the widget routine to exit with an error.

---

## Managing Widget Application State

Usually, a widget application or compound widget has some information, or *state*, associated with it. This is a natural consequence of the fact that the application is usually divided into at least two separate routines, one that creates and realizes the application and another that handles events. These multiple routines need shared access to certain types of information such as the widget IDs of the component widgets and data being used by the application.

One obvious answer to this problem is to use a COMMON block to hold the state. However, this solution is undesirable because it prevents more than a single copy of the application from running at the same time. It is easy to imagine the chaos that would ensue if multiple instances of the same application were using the *same* common block without some sort of interlocking.

A better solution to this problem is to use the user value of one of the widgets to store state information for the application. Since this user value can be of any type, a structure can be used to store any number of state-related values. Using this technique, multiple instances of the same widget code can exist simultaneously.

In our previous discussions, the HANDLER field of widget event structures was described without giving any compelling reason for its existence. That is because event processing and compound widgets must be understood before the need for HANDLER becomes clear. Recall that when WIDGET\_EVENT finds an event to return, it moves up the widget tree looking for an event-handling routine registered to the widgets in between its current position and the top-level base of the widget application. If such a routine is found, it is called with the event as its argument, and the HANDLER field of this event is set to the widget ID of the widget where the event routine was found. Since compound widgets have event handlers associated with their root widget, the HANDLER field gives the event handler the widget ID of the root widget. This allows the event handler for a compound widget instance to easily locate the location of its state information relative to this root.

IDL programmers are often tempted to store the state information directly in the user value of the root widget, but this is not a good idea. The user value of a compound widget is reserved for the user of the widget, just like any basic widget. Therefore, you should store the state information in the user value of one of the child widgets below the root. As a convention, the user value of the first child is often used, leading to event handlers structured as follows:

```
FUNCTION EVENT_FUNC, event
; Get state from the first child of the compound widget root:
child = WIDGET_INFO(event.handler, /CHILD)
```



```
WIDGET_CONTROL, child, GET_UVALUE=state, /NO_COPY

; Execute event-handling code here.

; Restore the state information before exiting routine:
WIDGET_CONTROL, child, SET_UVALUE=state, /NO_COPY

; Return result of function
RETURN, result
END
```

Notice the use of the `NO_COPY` keyword in the above example. This keyword behaves similarly to the `TEMPORARY` function, and prevents IDL from duplicating the memory used by the user value during the `GET_UVALUE` and `SET_UVALUE` operations. This is an important efficiency consideration if your code generates many events or the size of the state data is large.

Sometimes, an application will find that it needs to use the user value of all its child widgets for some other purpose, and there is no convenient place to keep the state information. One way to work around this problem is to interpose an extra base between the root base and the rest of the widgets:

```
ROOT = WIDGET_BASE(parent)
EXTRA = WIDGET_BASE(root)
```

In such an approach, the remaining widgets would all be children of `EXTRA` rather than `ROOT`.

# Compound Widgets

Widget primitives can be used to construct many varied user interfaces, but complex programs written with them suffer the following drawbacks:

- Large widget applications become difficult to maintain. As an application grows, it becomes more difficult to properly write and test. The resulting program suffers from poor organization.
- Good ideas can be difficult to reuse. Most larger applications are constructed from smaller sub-units. For example, a color table editor might contain control panel, color selection and color-index selection sub-units. These sub-units are often complicated tools that could be used profitably in other programs. To reuse such sub-units, the programmer must understand the existing application and then transplant the interesting parts into the new program — at best a tedious and error-prone proposition.

*Compound widgets* solve these problems. A compound widget is a complete, self-contained, reusable widget sub-tree that behaves to a large degree just like a primitive widget. Complex widget applications written with compound widgets are much easier to maintain than the same application written without them. Using compound widgets is analogous to using subroutines and functions in programming languages.

## Writing Compound Widgets

Compound widgets are written in the same way as any other widget application. They are distinguished from regular widget applications in the following ways:

- Compound widgets usually have a base widget at the root of their hierarchies. This base contains the subwidgets that make up the cluster. From the user's point of view, this single widget *is* the compound widget — its children are hidden from the users view.
- It is important that the compound widget not make use of the base's user value. This user value should be reserved for use by the caller of the compound widget in order to preserve the illusion that the compound widget works just like any of the basic widgets.
- The root widget of the compound widget *always* has an event handler function associated with it via the `EVENT_FUNC` keyword to the widget creating function or the `WIDGET_CONTROL` procedure. This event handler manages events from its sub-widgets and generates events for the compound widget. By swallowing events from the widgets that comprise the compound widget and

generating events that represent the compound widget, it presents the illusion that the compound widget is acting like a basic widget.

- If the compound widget has a value that can be set, it should be assigned a value setting procedure via the `PRO_SET_VALUE` keyword to the widget creating function or the `WIDGET_CONTROL` procedure.
- If the compound widget has a value that can be retrieved, it should be assigned a value retrieving function via the `FUNC_GET_VALUE` keyword to the widget creating function or the `WIDGET_CONTROL` procedure.

For an example of how a compound widget might be written, see [“Compound Widget Example”](#) on page 598.

# Tips on Creating Widget Applications

The following are some ideas to keep in mind when writing widget applications in IDL.

- When writing new applications, decompose the problem into sub-problems and write reusable compound widgets to implement them. In this way, you will build a collection of reusable widget solutions to general problems instead of hard-to-modify, monolithic programs.
- Use the `GROUP_LEADER` keyword to `WIDGET_BASE` to define the relationships between parts of your application. Group leadership/membership relationships make it easy to group widgets appropriately for iconization, layering, and destruction.
- Use the `MBAR` (and `APP_MBAR`) keyword to `WIDGET_BASE` to create application-specific menubars. Use keyboard focus events to track which widget menu options should affect.
- Use existing compound widgets when possible. In particular, use the `CW_BGROU`P and `CW_PDMENU` compound widgets to create menus. These functions are easier to use than writing the menu code directly, and your intent will be more quickly understood by others reading your code.
- The many advantages of the `XMANAGER` procedure dictate that all widget programs should use it. There are few if any reasons to call the `WIDGET_EVENT` procedure directly.
- Use `CATCH` to handle any unanticipated errors. The `CATCH` branch can free any pointers, pixmaps, logical units, etc., to which the calling routine will not have access, and reset IDL session-wide settings like color tables and system variables.
- If all else fails, it is possible to use the value of the `WIDGET_INFO` function to execute special-case code for each platform's user interface toolkit. It is desirable, however, to avoid large-scale special-case programming because this makes maintenance of the finished program more difficult.

## Portability Issues

Although IDL widgets are essentially the same on all supported platforms, there are some differences that can complicate writing applications that work well everywhere. The following hints should help you write such applications:

- Avoid specifying the absolute size and location of widgets whenever possible. (That is, avoid using the `XSIZE`, `YSIZE`, `XOFFSET`, and `YOFFSET` keywords.) The different user interface toolkits used by different platforms create widgets with slightly different sizes and layouts, so it is best to use bases that order their child widgets in rows or columns and stay away from explicit positioning. If you must use these keywords, try to isolate the affected widgets in a sub-base of the overall widget hierarchy to minimize the overall effect.
- When using a bitmap to specify button labels, be aware that some toolkits prefer certain sizes and give sub-optimal results with others. Also, if you are specifying a color bitmap, use the `BITMAP` keyword.
- Try to place text, label, and list widgets in locations where their absolute size can vary without making the overall application look bad. The font used by the different toolkits have different physical sizes that can cause these widgets to have different proportions.

It is reasonably easy to write applications that will work in all environments without having to resort to much special-case programming. It is very helpful to have a machine running each environment available so that the design can be tested on each iteratively until a suitable layout is obtained.

# Compound Widget Example

The following example incorporates ideas from the previous sections to show how you might approach the task of writing a compound widget. The widget is called `CW_DICE`, and it simulates a single six-sided die. [Figure 18-7](#) shows the appearance of `XDICE`, an application that uses two instances of `CW_DICE`. `XDICE` is discussed on [“Using `CW\_DICE` in a Widget Program”](#) on page 604.

---

**Note**

`cw_dice.pro` can be found in the `lib` subdirectory of the IDL distribution. `xdice.pro` can be found in the `doc` subdirectory of the examples subdirectory of the IDL distribution. You should examine these files for additional details and comments not included here. We present sections of the code here for didactic purposes—there is no need to re-create either of these files yourself.

---

The `CW_DICE` compound widget has the following features:

- It uses a button widget. The current value of the die is displayed as a bitmap label on the button itself. When the user presses the button, the die “rolls” itself by displaying a sequence of bitmaps and then settles on a final value. An event is generated that returns this final value.
- Timer events are used to create the rolling effect. This allows the dice to give the same appearance on machines of varying performance levels.
- The die can be set to a specific value via the `SET_VALUE` keyword to the `WIDGET_CONTROL` procedure. If the desired value is outside of the range 1 through 6, the die is rolled as if the user had pressed the button and a final value is selected randomly. Using `WIDGET_CONTROL` does not cause an event to be issued. This follows the IDL convention that user actions cause events while programmatic changes do not.
- The current value of the die can be obtained via the `GET_VALUE` keyword to the `WIDGET_CONTROL` procedure.

Almost any compound widget will have some state associated with it. The following is the state of `CW_DICE`:

1. The current value.
2. The number of times the die should “tumble” before settling on a final value.
3. The amount of time to take between tumbles.

4. When a roll is in progress, a count of how many tumbles are left before a final value is displayed.
5. The bitmaps to use for the 6 possible die values.
6. The seed to use for the random number generator.

The first four items are stored in a per-widget structure kept in one of the child widget's user values. Since the bitmaps never change, it makes sense to keep them in a COMMON block to be accessed freely by all the CW\_DICE routines. It also makes sense to use a single random number seed for the entire CW\_DICE class rather than one per instance to avoid the situation where multiple dice, having been created at the same time, have the same seed and thus display the same value on each roll.

It is rare that the use of a COMMON block in a compound widget makes sense. Notice, however, that we're only keeping read-only data (bitmaps) or data that can be overwritten at any time with no negative effects (random number generator seed).

Given the above decisions, it is now possible to write the CW\_DICE procedure:

```

;Value is an optional argument that lets the caller set the initial
;die value to a value between 1 and 6. UVALUE will simply be passed
;on to the root base of CW_DICE. The TUMBLE keywords let the user
;adjust the tumble count and period.
PRO cw_dice, parent, value, UVALUE=uvalue, $
    TUMBLE_CNT=tumble_cnt, TUMBLE_PERIOD=tumble_period

;This COMMON block holds the bitmaps and random number generator
;seed.
COMMON CW_DICE_BLK, seed, faces

;Provide defaults for the keywords.
IF NOT KEYWORD_SET(tumble_cnt) THEN tumble_cnt=10

;Guard against a nonsensical request.
IF (tumble_cnt lt 1) then tumble_cnt=10

;Default tumble period in seconds.
IF NOT KEYWORD_SET(tumble_period) THEN tumble_period=.05
IF (tumble_period lt 0) then tumble_period=.05
IF NOT KEYWORD_SET(uvalue) uvalue=0

;Return to caller if an error occurs.
ON_ERROR, 2

;Generate the die face bitmaps. The actual code for this is omitted
;here because it doesn't add much to the example, but it can be

```

```

;found in the CW_DICE.PRO file.
faces=LONARR(192)

;Use RANDOMU to pick the initial value of the die unless the user
;provided one.
IF(N_ELEMENTS(value) EQ 0) THEN value = FIX(6*RANDOMU(seed) + 1)

;Construct a state variable for this instance.
state = { value:value, tumble_cnt:FIX(tumble_cnt), $
          tumble_period:tumble_period, remaining:0 }

;Create the base widget, passing the UVALUE through for the caller.
;Notice that we also register an event function and GET/SET value
;routines which will be called by WIDGET_CONTROL on our behalf.
base = WIDGET_BASE(parent, UVALUE=uvalue, $
          EVENT_FUNC='CW_DICE_EVENT', $
          FUNC_GET_VALUE='CW_DICE_GET_VALUE', $
          PRO_SET_VALUE='CW_DICE_SET_VALUE' )

;Create the die, setting its bitmap to the current value.
die = WIDGET_BUTTON(base, VALUE=faces[*], *, value-1)

;Save the state in the first child's user value. Notice the use of
;the NO_COPY keyword for efficiency.
WIDGET_CONTROL, WIDGET_INFO(base, /CHILD), $
  SET_UVALUE=state, /NO_COPY

;The result of a compound widget is always the ID of its topmost
;widget.
RETURN, base

END

```

The above code makes reference to two routines named `CW_DICE_SET_VAL` and `CW_DICE_GET_VAL`. By using the `FUNC_GET_VALUE` and `PRO_SET_VALUE` keywords to `WIDGET_BASE`, `WIDGET_CONTROL` can call these routines whenever the user makes a `WIDGET_CONTROL SET_VALUE` or `GET_VALUE` request:

```

;This is the SET_VALUE routine for CW_DICE. The number and type of
;the arguments is defined by WIDGET_CONTROL. Id is the widget ID of
;a CW_DICE, and value is the user's requested value.
PRO cw_dice_set_val, id, value

COMMON CW_DICE_BLK, seed, faces

;Get the ID of the first child of the CW_DICE widget. This is where
;the state information is stored.
stash = WIDGET_INFO(id, /CHILD)

```



```

;Get the state structure.
WIDGET_CONTROL, stash, GET_UVALUE=state, /NOCOPY

;If the value is outside the range [1,6] then roll the die as if
;the user pressed the button.
if (value < 1) or (value > 6) THEN BEGIN

;CW_DICE_ROLL rolls the dice. It's a separate function because our
;event handler also needs to use it.
CW_DICE_ROLL, stash, state

ENDIF ELSE BEGIN
;If the value is in the range [1,6] then simply set the die to that
;value without rolling.
state.value=value

;Set the new bitmap on the button. We take advantage of the fact
;that stash must be the widget ID of the button widget, since the
;base only has one child.
WIDGET_CONTROL, stash, SET_VALUE=faces[*,*, value-1]

ENDELSE

;Restore the state in the child UVALUE.
WIDGET_CONTROL, stash, SET_UVALUE=state, /NO_COPY

END

;This is the GET_VALUE routine for CW_DICE. The number and type of
;the arguments is defined by WIDGET_CONTROL. Id is the widget ID of
;a CW_DICE. The return value of this function must be the current
;value of the compound widget, as defined by that widget.
FUNCTION cw_dice_get_val, id

;Get the ID of the first child of the CW_DICE widget. This is where
;the state information is stored.
stash = WIDGET_INFO(id, /CHILD)

;Get the state structure.
WIDGET_CONTROL, stash, GET_UVALUE=state, /NO_COPY

;Get the current value from the state structure.
ret = state.value

;Restore the state in the child UVALUE.
WIDGET_CONTROL, stash, SET_UVALUE=state, /NO_COPY

```

```
RETURN, ret
```

```
END
```

CW\_DICE\_SET\_VALUE makes reference to a procedure named CW\_DICE\_ROLL that does the actual dice rolling. Rolling is implemented as follows:

1. If this is the initial call to CW\_DICE\_ROLL, then pick the final value that will end up being displayed and enter this into the widget's state. Hence, WIDGET\_CONTROL, /GET\_VALUE reports the final value instead of one of the intermediate "tumble" values no matter when it is called.
2. If this is not the final tumble, pick a random intermediate value and display that. Then, make another timer event request for the next tumble.
3. If this is the final tumble, use the saved final value.
4. CW\_DICE\_ROLL works in cooperation with the event handler function for CW\_DICE. Each timer event causes the event handler to be called and the event handler in turn calls CW\_DICE\_ROLL to process the next tumble.

```
;Roll the specified die. Dice is the widget ID of the button
;holding the bitmap, and state is the state as extracted from the
;CW_DICE UVALUE by the caller.
PRO cw_dice_roll, dice, state
COMMON CW_DICE_BLK, seed, faces

;First time.
IF (state.remaining EQ 0) THEN BEGIN

;Set the counter for the number of tumbles remaining.
state.remaining = state.tumble_cnt

;Determine final value now.
state.value = FIX(6*RANDOMU(seed)+1)

ENDIF

;Last time.
IF (state.remaining EQ 1) THEN BEGIN

;Use the previously-saved final result.
value = state.value

;Not the last time.
ENDIF ELSE BEGIN

;Generate an intermediate value.
```

```

value = FIX(6 * RANDOMU(seed) + 1)

;Since this isn't the last tumble, make the next timer request.
WIDGET_CONTROL, dice, TIMER=state.tumble_period

ENDELSE

;Display the correct bitmap.
WIDGET_CONTROL, dice, SET_VALUE=faces[*,*, value-1]

;Decrement tumble counter.
state.remaining = state.remaining-1

END

```

This leads us to the event handler function:

```

FUNCTION cw_dice_event, event

;The primary use for the HANDLER field of event structures is to
;make finding the root of a compound widget easy.
base = event.handler

;Get the ID of the first child of the CW_DICE widget. This is where
;the state information is stored.
stash = WIDGET_INFO(base, /CHILD)

;Get the state structure.
WIDGET_CONTROL, stash, GET_UVALUE=state, /NO_COPY

;Roll the die and display a new bitmap.
CW_DICE_ROLL, stash, state

;This event handler expects to see button press events generated
;from a user action as well as TIMER events from CW_DICE_ROLL. We
;only want to issue events for the button presses. Even though the
;die still has several tumbles left, we know that the final value
;is in the state now.
if (TAG_NAMES(event, /STRUCTURE_NAME) NE 'WIDGET_TIMER') THEN $

;Create an event.
ret = { CW_DICE_EVENT, ID:base, TOP:event.top, $
        HANDLER:OL, VALUE:state.value} $
ELSE ret = 0
;By not returning an event structure, we cause the event to be
;swallowed by WIDGET_EVENT.

;Restore the state in the child UVALUE.
WIDGET_CONTROL, stash, SET_UVALUE=state, /NO_COPY

```

```
RETURN, ret
```

```
END
```

This results in the following:

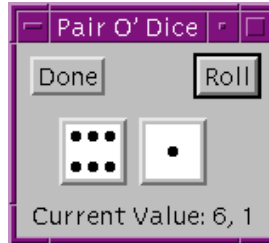


Figure 18-7: The XDICE Example Program

## Using CW\_DICE in a Widget Program

Having written a compound widget, it is natural to want to use it in a real application. We can use CW\_DICE to implement an application named XDICE. XDICE displays two dice as well as a “Roll” button. Pressing either die causes it to roll individually. Pressing the “Roll” button causes both dice to roll together. A text widget at the bottom always displays the current value in textual form. XDICE is shown in the preceding figure.

### Note

`xdice.pro` can be found in the `doc` subdirectory of the `examples` subdirectory of the IDL distribution. You can run the program from the IDL distribution by entering:

```
xdice
```

at the IDL command prompt. See [“Running the Example Code”](#) on page 543 if IDL does not run the program as expected. You should examine the files for additional details and comments not included here.

```
;Providing standard keywords usually found in other widget
;applications is a nice finishing touch. GROUP is easy to support
;since we just pass it to XMANAGER.
PRO xdice, GROUP=group
```

```

;Create the top-level base that holds everything else.
base = WIDGET_BASE(/COLUMN, title='Pair O'' Dice')

;A button group compound widget is used to implement the Done and
;Roll buttons. The SPACE keyword simply causes the buttons to be
;spread out from each other.
bgroup = CW_BGROU(bgroup, ['Done', 'Roll'], /ROW, SPACE=50)

;Create a row base to hold the dice. XPAD moves the first die away
;from the left side of the application and helps center the dice.
dice = WIDGET_BASE(base, /ROW, XPAD=20)

;The first die.
d1 = CW_DICE(dice)

;The second die.
d2 = CW_DICE(dice)

;We need the initial dice values to set the label appropriately. We
;could have specified initial values for the calls to CW_DICE
;above, but it seems better to let them be different on each
;invocation.
WIDGET_CONTROL, d1, GET_VALUE=d1v
WIDGET_CONTROL, d2, GET_VALUE=d2v

;Format the initial label text.
str=STRING(FORMAT='("Current Value: ",I1,", ",I1)', d1v, d2v)

;This label is used to textually display the current dice values.
label = WIDGET_LABEL(base, value=str)

;Information that is needed in the event handler.
state = { bgroup:bgroup, d1:d1, d2:d2, label:label }

;Save useful information in the base UVALUE, and realize the
;application.
WIDGET_CONTROL, base, SET_UVALUE=state, /NO_COPY, /REALIZE

;Pass control to XMANAGER.
XMANAGER, 'XDICE', base, GROUP=group

END

```

The following event handler is called by XMANAGER to process events for the XDICE application:

```

PRO xdice_event, event

;Recover the state.

```

```
WIDGET_CONTROL, event.top, GET_UVALUE=state, /NO_COPY

;Either the Done or Roll button was pressed.
IF (event.id EQ state.bgroup) THEN BEGIN

;The Done button.
IF (event.value EQ 0) THEN BEGIN

WIDGET_CONTROL, /DESTROY, event.top;Destroy the application.

;Return now to avoid trying to update the widget label we
;just destroyed.
RETURN

;The Roll button.
ENDIF ELSE BEGIN

;Roll the first die by asking for an out of range value.
WIDGET_CONTROL, state.d1, SET_VALUE=-1

;Roll the second die.
WIDGET_CONTROL, state.d2, SET_VALUE=-1

ENDELSE
ENDIF

;Get value of first die.
WIDGET_CONTROL, state.d1, GET_VALUE=d1v

;Get value of second die.
WIDGET_CONTROL, state.d2, GET_VALUE=d2v

;Format the initial label text.
str = STRING(format='("Current Value: ",I1,", ",I1)', d1v, d2v)

;Update the label.
WIDGET_CONTROL, state.label, SET_VALUE=str

;Restore the state.
WIDGET_CONTROL, event.top, SET_UVALUE=state, /NO_COPY

END
```



## Chapter 19:

# Debugging an IDL Program

The following topics are covered in this chapter:

---

Overview .....	608	The IDL Code Profiler .....	614
Debugging Commands .....	609	The Variable Watch Window .....	619

## Overview

There are several tools you can use to help you find errors in your IDL code. The Run menu item in the IDL Development Environment provides several ways to access IDL's built-in debugging and executive commands. The IDL Profiler provides useful information about the routines used in the program being executed. The Variable Watch Window helps you keep track of the variables used in your program.

This chapter explains the debugging commands and contains short examples using the IDLDE interface to debug a file.



# Debugging Commands

When a file displayed in an IDL editor window has been compiled (by selecting “Compile” or “Memory Compile” from the Run menu, or by entering `.COMPILE`, `.COMPILE -f`, or `.RUN` at the IDL command prompt), a number of debugging commands become available for selection. For more information on the Run menu, see the *Using IDL* manual.

When execution is interrupted, a current-line indicator is placed next to the line that will be executed when processing resumes. The routine being compiled need not already be shown in an editor window. If a routine compiled with the `.RUN`, `.RNEW`, or `.COMPILE` executive commands contains an error, IDLDE will display the file automatically.

## A Simple Example

A simple procedure, called `BROKEN`, has been included in the IDL distribution. An error occurs when `BROKEN` is executed.

Start the IDLDE. Call the `BROKEN` procedure by entering:

```
BROKEN
```

at the IDL command line. An error is reported in the Output Log window and an editor window containing the file `BROKEN.PRO` appears.

A “Variable is undefined” error has occurred. Since execution stopped at line 4, that line is highlighted with an arrow.

Click on the Output Log window to see the error:

```
% Compiled module: BROKEN.
% PRINT: Variable is undefined: I.
% Execution halted at BROKEN      4
  /user/local/rsi/idl50/examples/general/broken.pro
%                               $MAIN$
```

There are several ways of fixing this error. We could edit the program file to explicitly define the variable `i`, or we could change the program so that it accepts a parameter at the command line. We can also define the variable `i` “on the fly” and continue execution of the program without making any changes to the program file. We’ll do this first, then go back and edit the program to accept a command-line parameter.

To define the variable `i` and assign it the value 10, click in the IDL command line and enter:

```
i = 10
```

## Step Through the Program

Select “Step Into” from the Run menu to execute line 4 with the new value of `i` and step to the next program line.

The Output Log reports:

```
10
```

and the current-line pointer advances to the next line in the window containing the file `BROKEN.PRO`. You could continue stepping through the program by choosing “Step Into” repeatedly (or by entering `.STEP` at the IDL command prompt).

The Trace dialog offers an opportunity to automatically step through the program. Select “Trace...” from the Run menu. The Trace dialog appears. Click “Run” to continue issuing the `.STEP` command until the `END` statement is encountered. Click “Cancel” to dismiss the Trace dialog.

You can also continue execution of the program without stepping through. Select “Run” from the Run menu, noting that the Output Log shows that IDL calls `broken`. Define the variable `i` in the Command Input Line. Select “Run” again. The Output Log now shows that IDL calls `.CONTINUE`. IDL prints the resulting output to the Output Log window:

```
10  
20  
30  
40
```

When stepping through a main program, if the next line calls another IDL procedure or function, you have three options with which to handle execution of the nested program. Selecting “Step Into” executes statements in order by successive “Step” commands. Selecting “Step Over” executes statements to the end of the called function, without interactive capability. Select “Step Out” if you would like to continue processing until the main program returns.

## Fix the Program

To fix the program permanently, edit the first line of the program to be:

```
PRO BROKEN, i
```

Select “Save” from the File menu and “Compile” from the Run menu. IDL saves the modified text file over the old version and compiles the modified routine. To call this new version of BROKEN with an input argument of 10, enter:

```
BROKEN, 10
```

The Output Log window prints the result:

```
10
20
30
40
```

## Breakpoints

You can suspend execution of a program temporarily by setting breakpoints in the code. Set a breakpoint at the fifth line of BROKEN.PRO by placing the cursor in the line that reads:

```
PRINT, i*2
```

and selecting “Set Breakpoint” from the Run menu. A breakpoint dot appears next to the line. Now enter:

```
BROKEN, 10
```

The Output Log window displays the following:

```
10
% Breakpoint at: BROKEN          5
   user/local/rsi/idl40/examples/general/broken.pro
```

and a current-line indicator marks line 5. Select “Run” to allow execution to resume. To list the breakpoints, enter HELP/BREAKPOINT at the Command Input Line.

Setting a breakpoint allows you to inspect (or change) variable definitions as the program executes. Since our example does not set any variables, setting a breakpoint in BROKEN.PRO is not very informative. Breakpoints can be extremely helpful, though, when debugging complex programs, or programs that call other routines.

## The Breakpoint Tool Bar Buttons

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



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

Toggle Breakpoint button creates the breakpoint. If a breakpoint already exists in that line, the breakpoint is removed.



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



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

## The Edit Breakpoints Dialog

The **Edit Breakpoints** dialog allows you to add, remove, and remove all breakpoints in a file as well as the ability to move to the line in the source file that contains the breakpoint. The following figure shows the **Edit Breakpoints** dialog:

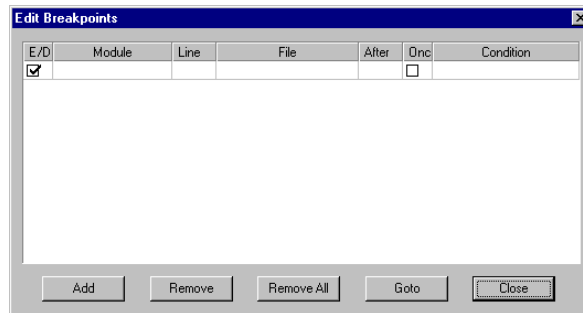



Figure 19-1: Edit Breakpoints Dialog

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

1. Open the file you in which you want to set a breakpoint.
2. Display the **Edit Breakpoints** dialog by clicking the  button in the IDLDE Tool Bar or by selecting **Run** → **Edit Breakpoints...**
3. Place the cursor in the line in which you want to create the breakpoint in the Editor window.

4. Select **Add** in the **Edit Breakpoints** dialog box. You will see a new entry display in the dialog. The following table describes each property of a breakpoint:

Item	Description
E/D	Specifies whether a breakpoint is enabled or disabled. If a check mark is displayed, the breakpoint is enabled and execution will stop when the all criteria for the breakpoint is met.
Module	Specifies the procedure or function the breakpoint is set in. <b>Note</b> - This item will not be displayed until the file has been compiled with the new breakpoint.
Line	Specifies the line number where breakpoint has been set.
File	Specifies the filename where the breakpoint has been set.
After	Specifies how many times the execution must pass the breakpoint before stopping execution. For example, if this item is set to 0, execution will stop the first time this breakpoint is encountered. If it is set to 9, execution will not stop until the breakpoint has been encountered for the ninth time.
Once	The breakpoint is removed after it is encountered for the first time.
Condition	Specifies a condition to be met for the execution to stop. The condition is a string containing an IDL expression. When a breakpoint is encountered, the expression is evaluated. If the expression is true (if it returns a non-zero value), program execution is interrupted. The expression is evaluated in the context of the program containing the breakpoint.

*Table 19-1: Edit Breakpoints Description*

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

Your breakpoint entry is now complete.

# The IDL Code Profiler

The IDL Code Profiler helps you analyze the performance of your applications. You can easily monitor the calling frequency and execution time for procedures and functions. The Profiler can be used with programs entered from the command line as well as programs run from within a file.

You can start the IDL Code Profiler by selecting “Profile” from the Run menu of the IDLDE or by entering `PROFILER` at the Command Input Line. For more information about the `PROFILER` procedure, see [PROFILER](#) in the *IDL Reference Guide*.

## Note

Calling the Profiler from the Command Input Line does not start the Profiler dialog.

## The Profile Dialog

Select “Profile” from the Run menu. The Profile dialog appears.

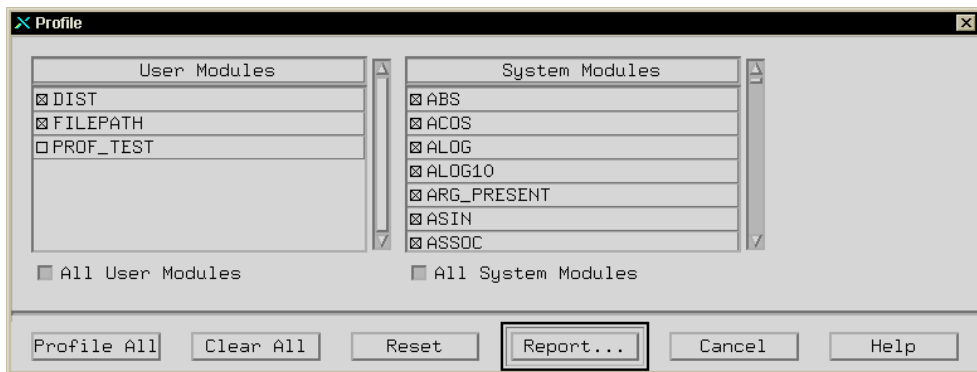


Figure 19-2: Profile Dialog

## User Modules

User modules include user-written procedures as well as library procedures and functions provided with IDL. By default, none of the User Modules are selected for profiling. To select a module, click on the checkbox next to it. All user modules must be compiled before opening the Profile dialog in order to be available for profiling.

### All User Modules

Select this checkbox to select all the user modules for profiling.

### System Modules

This field includes all IDL system procedures and functions.

### All System Modules

Select this checkbox to select all the system modules for profiling.

### Buttons

Click “Profile All” to enable profiling for all the available modules—System and User. Click “Clear All” to disable profiling for all the available modules—System and User. Click “Reset” to clear the report shown in the “Profile Report” dialog. The “Profile Report” dialog is dismissed, as it no longer contains any information. Click “Report” to generate a profile of the selected modules. The Profile Report dialog appears. Click “Cancel” to dismiss the Profile dialog. Click “Help” to display Help on this dialog.

## The Profile Report Dialog

Click “Report” from the Profile dialog in the Run menu of the IDLDE. The Profile Report dialog appears.

### Fields in the Profiler Report Dialog

The fields in the Profiler Report dialog show the following attributes of the modules selected for profiling from the Profile dialog. You can sort the values in each column in both ascending and descending order by clicking anywhere within the column. By default, the Modules column is sorted alphabetically.

#### Note

---

Whether you enter a program at the command line, or run a program contained in a file, the PROFILER procedure will report the status of all the specified modules compiled and executed either since profiling was first set or since the PROFILER was reset.

---

### Modules

The name of the library, user, or system procedure or function.

**Typ**

The type of module. System procedures or functions are associated with an “S”. User or library functions or procedures are associated with a “U”.

**Count**

The number of times the procedure or function has been called.

**Only(sec)**

The time required, in seconds, for IDL to execute the given function or procedure, not including any calls to other functions or procedures (children).

**Only Avg**

Average of the Only(sec) field above.

**+Children(sec)**

The time required, in seconds, for IDL to execute the given function or procedure including any calls to other functions or procedures.

**+Child Avg**

Average of the +Children(sec) field above.

**Buttons**

Click “Print” to print the report. The Print dialog appears. You can also select “Print” from the File menu of the IDLDE. Click “Save” to save the report as a text file. The Save Profile Report dialog appears. Click “Cancel” to dismiss the Profile Report dialog. The contents remain available after cancelling. Click “Help” to display Help on this dialog.

**Using the IDL Code Profiler**

Open a new editor file by selecting “New” from the File menu.

Enter the following lines in the editor:

```

pro prof_test
  openr, 1, filepath('nyny.dat', subdir=['examples', 'data'])
  a=assoc(1, bytarr(768,512))
  b=a[0]
  close, 1
  TV, b
end

```

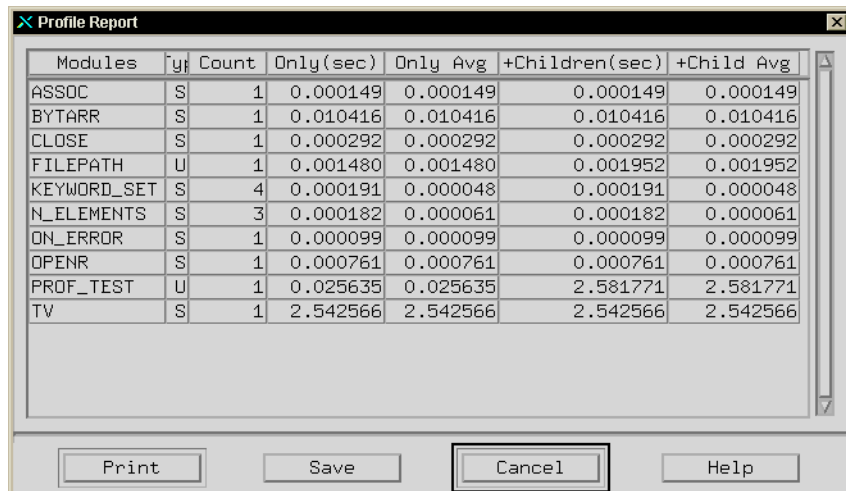


Save the file as `prof_test.pro` by selecting “Save” from the File menu. The Save As dialog appears.

To use the IDL Code Profiler, you must first compile the routines you would like to profile. For more involved programs, you can use `RESOLVE_ALL` to compile all uncompiled functions or procedures that are called in any already-compiled procedure or function.

Select “Profile...” from the Run menu. The Profile dialog appears; it will remain visible until dismissed. Select “Profile All” to profile all the available modules.

Run the application by selecting “Run” from the File menu. After the application is finished, return to the Profile dialog and click “Report”. The Profile Report dialog appears, as shown in the following figure.



Modules	Type	Count	Only(sec)	Only Avg	+Children(sec)	+Child Avg
ASSOC	S	1	0.000149	0.000149	0.000149	0.000149
BYTARR	S	1	0.010416	0.010416	0.010416	0.010416
CLOSE	S	1	0.000292	0.000292	0.000292	0.000292
FILEPATH	U	1	0.001480	0.001480	0.001952	0.001952
KEYWORD_SET	S	4	0.000191	0.000048	0.000191	0.000048
N_ELEMENTS	S	3	0.000182	0.000061	0.000182	0.000061
ON_ERROR	S	1	0.000099	0.000099	0.000099	0.000099
OPENR	S	1	0.000761	0.000761	0.000761	0.000761
PROF_TEST	U	1	0.025635	0.025635	2.581771	2.581771
TV	S	1	2.542566	2.542566	2.542566	2.542566

Figure 19-3: Profile Report Dialog

For more information about the capabilities of either dialog, see “The Profile Dialog” on page 614 and “The Profile Report Dialog” on page 615.

## Profiling with Command Line Modules

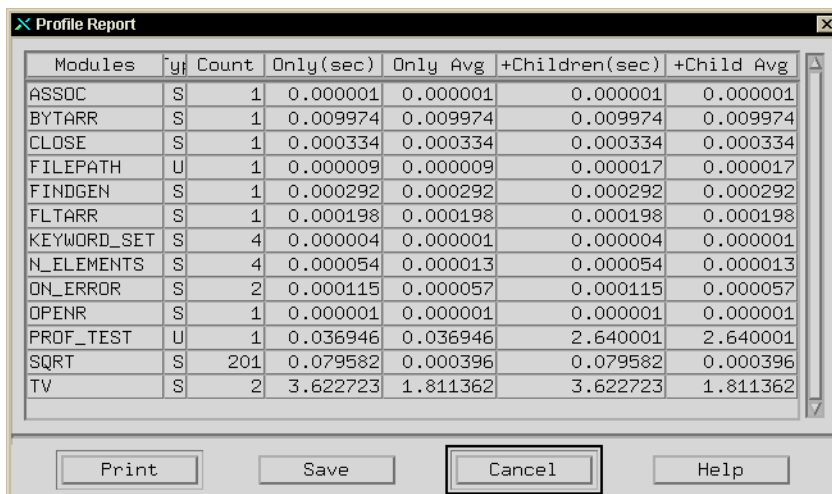
We will demonstrate how the Profiler handles newly compiled modules. The above example set profiling for all system files, plus the user module, `prof_test`, and the library function, `FILEPATH`. If you have altered the above results, reset the report and run `prof_test` again.

Enter the following lines at the Command Input Line:

```
;Create a dataset using the library function DIST. Note that DIST
;is immediately compiled.
A= DIST(500)

;Display the image.
TV, A
```

Return to the Profile dialog. You will note that the DIST function has been appended to the User Module field, but that it remains unselected. The Profiler will not include any uncompiled modules by default. Click “Report” in the Profile dialog to refresh the Profile Report dialog’s results. The following figure shows the new results. Note that TV is counted twice, and that more system modules have been appended to the Modules column. The DIST function, although it is not itself included, calls system routines which were previously selected for profiling.



Modules	Type	Count	Only(sec)	Only Avg	+Children(sec)	+Child Avg
ASSOC	S	1	0.000001	0.000001	0.000001	0.000001
BYTARR	S	1	0.009974	0.009974	0.009974	0.009974
CLOSE	S	1	0.000334	0.000334	0.000334	0.000334
FILEPATH	U	1	0.000009	0.000009	0.000017	0.000017
FINDGEN	S	1	0.000292	0.000292	0.000292	0.000292
FLTARR	S	1	0.000198	0.000198	0.000198	0.000198
KEYWORD_SET	S	4	0.000004	0.000001	0.000004	0.000001
N_ELEMENTS	S	4	0.000054	0.000013	0.000054	0.000013
DN_ERROR	S	2	0.000115	0.000057	0.000115	0.000057
OPENR	S	1	0.000001	0.000001	0.000001	0.000001
PROF_TEST	U	1	0.036946	0.036946	2.640001	2.640001
SQRT	S	201	0.079582	0.000396	0.079582	0.000396
TV	S	2	3.622723	1.811362	3.622723	1.811362

*Figure 19-4: Refreshing the Profile Report*

If you select DIST in the User Modules field in the Profile dialog and then re-enter only the statement calling TV at the Command Input Line, you will notice that only the count for TV increases in the profiler report. You must re-enter the statement calling DIST at the Command Input Line; the already-compiled library function is executed again, making it available for profiling.

# The Variable Watch Window

The Variable Watch window displays current variable values after IDL has completed execution. If the calling context changes during execution — as when stepping into a procedure or function — the variable table is replaced with a table appropriate to the new context. While IDL is at the main program level, the Watch window remains active and displays any variables created.

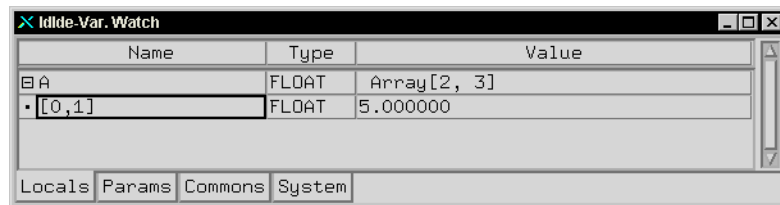


Figure 19-5: Variable Watch Window

## Customizing Variable Watch Window Layout

To hide the Variable Watch window, select “Hide Variable Watch” from the Configure option in the Window menu. Select “Show Variable Watch” to make it reappear. Changing the Window menu will only affect the current IDL session. To apply your changes to future sessions, go to the Layout tab from the Preferences option of the File menu. In the section labeled Windows, you can use the Hide field to make any of the available options disappear. Click “Save” to apply any changes to future IDL sessions.

### Note

The Configure option from the Window menu reflects changes in the Layout Preferences and vice versa.

You can also choose to separate the Variable Watch window from the main IDLDE window. Use the Separate field in the Layout Preferences tab from the File menu.

## The Variable Watch Interface Description

The Variable Watch window is refreshed after the IDLDE has completed execution. Each Variable Watch window contains the following folders:

- **Locals**

This tab contains descriptions of local variables. Local variables are created from IDL's main program level. For example, entering `a=1` at the Command Input Line lists the integer `a` in the Locals tab.
- **Params**

This tab contains descriptions of parameters. The variables and expressions passed to a function or procedure are parameters. For more information, see [“Parameters”](#) on page 286.
- **Commons**

This tab contains descriptions of variables contained in common blocks. The name of each common block is shown in parentheses next to the variable contained within it. For more information, see [“Common Blocks”](#) on page 208.
- **System**

This tab contains descriptions of system variables. System variables are a special class of predefined variables available to all program units. For more information about system variables, see [Appendix D, “System Variables”](#) in the *IDL Reference Guide*.

Each tab contains a table listing the variables included in the category. You can size the columns by clicking on the line to the right of the title of the column you wish to expand or shrink. Drag the mouse either left or right until you are satisfied with the width of the column. For example, to change the width of the Name column, click and drag on the line separating the Name field from the Type field.

The following fields describe variable attributes:

- **Name**

This field shows the name of the variable. This field is read-only, except for array subscript descriptions (see example in [Using the Variable Watch Window](#) below).

For compound variables such as arrays, structures, pointers, and objects, click the “+” symbol to the left of the name to show the variables included in the compound variable. Click the “-” symbol to collapse the description.
- **Type**

This field shows the type of the variable. This field is read-only.

- Value

This field shows the value of the variable. To edit a value, highlight the cell by clicking on it, press the function key F2 to enter editing mode, and enter the new value.

The Name, Type, and Value fields are displayed as when using the [HELP](#) procedure. For more information about variables, see “[Variables](#)” on page 96.

## The Variable Watch Window and Objects

Object references are expanded only if they reference non-null objects. Object data are expanded only if the object method has finished running. Object data are read-only and cannot be changed with the Variable Watch window.

## Using the Variable Watch Window

Arrays are expanded to show one array element. Click on the “+” symbol next the name of the array to display the initial array subscript. You can change this field to display the characteristics of any other array element. To edit the subscript, highlight the cell by clicking on it, press the function key F2 to enter editing mode, and modify the name using the arrow keys to maneuver. For example, enter the following:

```
;Create an array with 2 columns and 3 rows.
A=MAKE_ARRAY(2,3)

;Show the values of array A in the Output Log. They will all be
;zero.
PRINT, A

;Assign the value of 5 to the value in the array subscripted as 2.
;This is the same as entering A(0,1)=5.
A(2)=5

;Show the new values of array A.
PRINT, A
```

IDL prints:

```
0.00000      0.00000
5.00000      0.00000
0.00000      0.00000
```

It is easy to manipulate variables within the Watch window. Click on the “+” expansion bitmap next to the array A. The subscript [0,0] will be revealed beneath the description of A. Enter editing mode and change [0,0] to [0,1].

**Note**

---

To enter editing mode in Motif, press F2 after clicking on the cell to be edited. In Windows, double click on the cell. On the Macintosh, click on the cell.

---

Press Return [Enter] to effect the change. Notice that the value of the subscript is displayed as 5.00000, as you entered from the Command Input Line. Press the “Tab” key twice to highlight the value of the subscript [0,1]. You can change it to another number. Enter [1,0] in the subscript name field. You can change the value from 0.00000 to another number.

For more information about arrays, see [“Arrays”](#) on page 90.



# Chapter 20: Building Cross- Platform Applications

The following topics are covered in this chapter:

---

Overview .....	624	Display Characteristics and Palettes .....	635
Which Operating System is Running? ..	625	Fonts .....	636
File and Path Specifications .....	626	Printing .....	637
Environment Variables .....	629	SAVE and RESTORE .....	638
Files and I/O .....	630	Widgets .....	639
Math Exceptions .....	633	Using External Code .....	642
Operating System Access .....	634	IDL DataMiner Issues .....	643

## Overview

IDL is designed as a platform-independent environment for data analysis and programming. Because of this, the vast majority of IDL's routines operate the same way no matter what type of computer system you are using. IDL's cross-platform development environment makes it easy to develop an application on one type of system for use on any system IDL supports.

Despite IDL's cross-platform nature, there *are* differences between the computers that make up a multi-platform environment. Operating systems supply resources in different ways. While IDL attempts to abstract these differences and provide a common environment for all Windows, Macintosh, UNIX, and VMS machines, there are some cases where the discrepancies cannot be overcome. This chapter discusses aspects of IDL that you may wish to consider when developing an application that will run on multiple types of computer.

---

**Note**

This chapter is *not* an exhaustive list of differences between versions of IDL for different platforms. Rather, it covers issues you may encounter when writing cross-platform applications in IDL.

---



## Which Operating System is Running?

In some cases, in order to effectively take platform differences into account, your application will need to execute different code segments on different systems. Operating system and IDL version information is contained in the IDL system variable `!VERSION`. For example, you could use an IDL [CASE Statement](#) that looks something like the following to execute code that pertains to a particular operating system family:

```
CASE !VERSION.OS_FAMILY OF
  'MacOS'      : Code for Macintosh
  'unix'       : Code for Unix
  'vms'        : Code for VMS
  'Windows'   : Code for Windows
ENDCASE
```

Writing conditional IDL code based on platform information should be a last resort, used only if you cannot accomplish the same task in a platform-independent manner.

# File and Path Specifications

Different operating systems use different path specification syntax and directory separation characters. The following table summarizes the different characters used by different operating systems; see [!PATH](#) in the *IDL Reference Guide* for further details on path specification.

Operating System	Directory Separator	Path Element Separator
MacOs	: (colon)	, (comma)
UNIX	/ (forward slash)	: (colon)
VMS	. (dot)	, (comma)
Windows	\ (backward slash)	; (semicolon)

*Table 20-1: Directory and Path Element Separator Characters*

As a result of these differences, specifying filenames and paths explicitly in your IDL application can cause problems when moving your application to a different platform. You can effectively isolate your IDL programs from platform-specific file and path specification issues by using the [FILEPATH](#) and [DIALOG\\_PICKFILE](#) functions.

## Choosing Files at Runtime

To allow users of your application to choose a file at runtime, use the [DIALOG\\_PICKFILE](#) function. [DIALOG\\_PICKFILE](#) will always return the file path with the correct syntax for the current platform. Other methods (such as reading a file name from a text field in a widget program) may or may not provide a proper file path.

## Selecting Files Programmatically

To give your application access to a file you know to be installed on the host, use the [FILEPATH](#) function. By default, [FILEPATH](#) allows you to select files that are included in the IDL distribution tree. Chances are, however, that a file you supply as part of your

own application is *not* included in the IDL tree. You can still use `FILEPATH` by explicitly specifying the root of the directory tree to be searched.

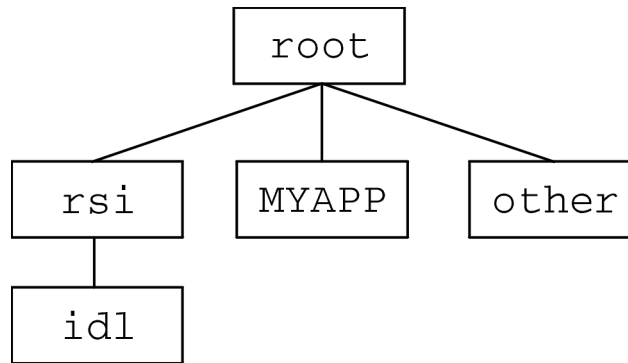


Figure 20-1: A possible directory hierarchy for an IDL application.

For example, suppose your application is installed in a subdirectory named `MYAPP` of the root directory of the filesystem that contains the IDL distribution. You could use the `FILEPATH` function and set the `ROOT_DIR` keyword to the root directory of the filesystem, and use the `SUBDIRECTORY` keyword to select the `MYAPP` directory. If you are looking for a file named `myapp.dat`, the `FILEPATH` command looks like this:

```
file = FILEPATH('myapp.dat', ROOT_DIR=root, SUBDIR='MYAPP')
```

The problem that remains is how to specify the value of `root` properly on each platform. This is one case where it is very difficult to avoid writing some platform-specific code. We could write an IDL `CASE` statement each time the `FILEPATH` function is used. Instead, the following code segment sets an IDL variable to the string value of the root of the filesystem, and passes that variable to the `ROOT_DIR` keyword. The `CASE` statement looks like this:

```
CASE !VERSION.OS_FAMILY OF
  'MacOS'   : rootdir = STRMID(!DIR, 0, STRPOS(!DIR, ':'))
  'unix'    : rootdir = '/'
  'vms'     : rootdir = 'SYS$SYSDEVICE:'
  'Windows' : rootdir = STRMID(!DIR, 0, 2)
ENDCASE
file = FILEPATH('myapp.dat', ROOT=rootdir, SUBDIR='MYAPP')
```

Note that the root directories under Unix and VMS are well defined, whereas the root directories on machines running the Macintosh OS or Microsoft Windows must be

determined by parsing the IDL system variable !DIR. On the Macintosh, the rootdir variable takes the value of !DIR up to the first directory separator character (a colon, in this case). On machines running Microsoft Windows, the root is assumed to be the drive letter of the hard drive and the following colon — usually “C:”.

# Environment Variables

UNIX and VMS versions of IDL have the ability to use *environment variables* (or *logical names*, under VMS) to store information about the environment in which IDL is running. Typically, environment variables are used to store information like the path to the main IDL directory, or to a batch file to be read and executed when IDL starts up. See “[Environment Variables Used by IDL](#)” in Chapter 2 of the *Using IDL* manual for details.

Microsoft Windows systems also have the ability to use environment variables to store information, but this form of information storage is much less common under Windows. On the Macintosh, there is no analogue of the environment variable.

Rather than using environment variables, the IDL Development Environment stores information in *preferences*; the mechanisms used to store preferences is different between platforms, but is generally transparent to you. Configuration settings you specify in the preferences dialogs of the IDL Development Environment are saved and are available to the IDE the next time it is started.

What does this all mean in the context of writing IDL applications for multiple platforms? Simply this: don't rely on environment variables in your programs unless you know that:

1. the target platform supports environment variables, and
2. the appropriate environment variables are defined as you wish them to be on the target platform.

## Files and I/O

IDL's file input and file output routines are designed to work identically on all platforms, where possible. In the case of basic operations, such as opening a text file and reading its contents, importing an image format file into an IDL array, or writing ASCII data to a file on a hard disk, IDL's I/O routines work the same way on all platforms. In more complicated cases, however, such as reading data stored in binary data format files, different operating systems may use files that are structured differently, and extra care may be necessary to ensure that IDL reads or writes files in the proper way.

Before attempting to write a cross-platform IDL application that uses more than basic file I/O, you should read and understand the sections in [Chapter 16, “Files and Input/Output”](#) that apply to the platforms your application will support. The following are a few topics to think about when writing IDL applications that do input/output.

### Byte Order Issues

Computer systems on which IDL runs support two ways of ordering the bytes that make up an arbitrary scalar: *big endian*, in which multiple byte numbers are stored in memory beginning with the most significant byte, and *little endian*, in which numbers are stored beginning with the least significant byte. The following table lists the processor types and operating systems IDL supports and their byte ordering schemes:

Processor Type	Operating System	Byte Ordering
Digital Alpha AXP	Tru64 UNIX	little-endian
	Alpha VMS	little-endian
	Windows NT	little-endian
Hewlett Packard PA-RISC	HP-UX	big-endian
IBM RS/6000	AIX	big-endian
Intel x86	Linux	little-endian
	Solaris x86	little-endian
	Windows	little-endian

Table 20-2: Byte ordering schemes used by platforms that support IDL

Processor Type	Operating System	Byte Ordering
Motorola PowerPC	Macintosh OS	big-endian
SGI R4000 and up	Irix	big-endian
Sun SPARC	SunOS	big-endian
	Solaris	big-endian

*Table 20-2: Byte ordering schemes used by platforms that support IDL*

The IDL routines [BYTEORDER](#) and [SWAP\\_ENDIAN](#) allow you to convert numbers from big endian format to little endian format and *vice versa*. It is often easier, however, to use the XDR (for eXternal Data Representation) format to store data that you know will be used by multiple platforms. XDR files write binary data in a standard “canonical” representation; as a result, the files are slightly larger than pure binary data files. XDR files can be read and written on any platform that supports IDL. XDR is discussed in detail in “[Portable Unformatted Input/Output](#)” on page 401.

## Logical Unit Numbers

Logical Unit Numbers (LUNs) are assigned to individual files when the files are opened by the IDL `OPENR/OPENU/OPENW` commands, and are used to specify which file IDL should read from or write to. There are a total of 128 LUNs available for assignment to files. While it is possible to assign any of the integers between 1-99 to a given file, when writing applications for others it is good programming practice to let IDL assign and manage the LUNs itself. By using the `GET_LUN` keyword to the `OPEN` routines, you can ask IDL to assign a free Logical Unit Number between 100-128 to the specified file. Letting IDL assign the LUN from the list of free unit numbers ensures that your application does not attempt to use a LUN already in use by someone else’s application. See the description of the `GET_LUN` keyword to [OPEN](#) in the *IDL Reference Guide* and “[Logical Unit Numbers \(LUNs\)](#)” on page 360.

## Macintosh File Pointer

IDL provides the [POINT\\_LUN](#) procedure to allow you to explicitly position the file pointer anywhere within an open file. Note, however, that on the Macintosh, the `POINT_LUN` routine cannot be used to position the file pointer past the end of the file, as it can on other platforms.

## Macintosh File Types and Creators

The Macintosh file system attaches two pieces of information to each file that is not used by other operating systems. The Macintosh file type specifies what type of data is stored in the file—for example, a file may contain text, an image, or unspecified binary information. The Macintosh file creator specifies which application created the file.

Text files saved by IDL on the Macintosh have the default file type “TEXT”. Binary files saved by IDL on the Macintosh have the default file type “BIN ” (note that the fourth character is a space). All files created by IDL have the default creator type “MIDL”. The default types can be overridden using the `MACCREATOR` and `MACTYPE` keywords to the `OPEN` routines. See [OPEN](#) in the *IDL Reference Guide* for details.

## Naming of IDL .pro Files

When naming IDL .pro files used in cross-platform applications, be aware of the various platforms’ file naming conventions and limitations. For example, the “\$” character is not allowed in a filename under VMS.

Be careful with case when naming files. For example, while Microsoft Windows systems present file names using mixed case, file names are in fact case-insensitive. File names are case-insensitive under VMS as well. Under Unix and the Macintosh operating system, file names are case sensitive—file.pro is different from File.pro. When writing cross-platform applications, you should avoid using filenames that are different only in case. The safest course is to use filenames that are all lower case.

Remember, too, that IDL commands are themselves case-insensitive. If entered at the IDL command prompt, the following are equivalent:

```
IDL> command
IDL> COMMAND
IDL> CommanD
```

One upshot of this is that if you have filenames that differ only in case and you use IDL’s automatic compilation feature, on platforms where case matters, IDL will look for the lower-case version of the file name first. You *can* specify case-sensitive filenames if you use the `.COMPILE` and `.RUN` executive commands—but again, we recommend that you use unique file names always.



# Math Exceptions

The detection of math errors, such as division by zero, overflow, and attempting to take the logarithm of a negative number, is hardware and operating system dependent. Some systems trap more errors than other systems. Beginning with version 5.1, IDL uses the IEEE floating-point standard on all supported systems. As a result, IDL always substitutes the special floating-point values NaN and Infinity when it detects a math error. (See [“Special Floating-Point Values”](#) on page 346 for details on NaN and Infinity.)

## Operating System Access

While IDL provides ways to interact with each operating system under which it runs, it is not generally useful to use operating-system native functions in a cross-platform IDL program. If you find that you must use operating-system native features, be sure to determine the current operating system (as described in [“Which Operating System is Running?”](#) on page 625) and branch your code accordingly.

# Display Characteristics and Palettes

## Finding Screen Size

Use the `GET_SCREEN_SIZE` function to determine the size of the screen on which your application is displayed. Writing code that checks the screen size allows your application to handle different screen sizes gracefully.

## Number of Colors Available

Use the `N_COLORS` and `TABLE_SIZE` fields of the `!D` system variable to determine the number of colors supported by the display and the number of color-table entries available, respectively.

Make sure that your application handles relatively small numbers of colors (less than 256, say) gracefully. For example, Microsoft Windows reserves the first 20 colors out of all the available colors for its own use. These colors are the ones used for title bars, window frames, window backgrounds, scroll bars, etc. If your application is running on a Windows machine with a 256-color display, it will have at most 236 colors available to work with.

Similarly, make sure that your application handles TrueColor (24-bit or 32-bit color) displays as well. If your application uses IDL's color tables, for example, you will need to force the application into 8-bit mode using the command

```
DEVICE, DECOMPOSED=0
```

to use indexed-color mode on a machine with a TrueColor display.

# Fonts

IDL uses three font systems for writing characters on the graphics device, whether that device be a display monitor or a printer: Hershey (vector) fonts, TrueType (outline) fonts, and device (hardware) fonts. Fonts are discussed in detail in [Appendix G, “Fonts”](#) in the *IDL Reference Guide*.

Both TrueType and Vector fonts are displayed identically on all of the platforms that support IDL. This means that if your cross-platform application uses either the TrueType fonts supplied with IDL or the Vector fonts, there is no need for platform-dependent code.

# Printing

IDL displays operating-system native dialogs using the `DIALOG_PRINTJOB` and `DIALOG_PRINTERSETUP` functions. Since the dialogs that control printing and printer setup differ between systems, so do the options and capabilities presented via IDL's print dialogs. If your IDL application uses IDL's printing dialogs, make sure that your interface calls the dialog your user will expect for the platform in question.

## SAVE and RESTORE

Unless your cross-platform application supports VMS, there are no platform-specific issues to be concerned with. However, if you distribute your application via IDL SAVE files, remember that files containing IDL routines are not necessarily compatible between IDL releases. Always save your original code and re-save when a new version of IDL is released. SAVE files containing data are always compatible between releases of IDL.

If your application supports VMS, you should be aware that SAVE files created on VMS machines with IDL versions before release 5.1 stored floating-point numbers in VAX format. Beginning with version 5.1, IDL stores all floating-point numbers in IEEE format. When IDL reads an older data file created on a VAX, it automatically converts the floating-point numbers from VAX format to IEEE format.

Note also that if you are restoring a file created with VAX IDL version 1, you must restore on a machine running VMS.

# Widgets

IDL's user interface toolkit is designed to provide a "native" look and feel to widget-based IDL applications. Where possible, widget toolkit elements are built around the operating system's native dialogs and controls; as a result, there are instances where the toolkit behaves differently from operating system to operating system. This section describes a number of platform-dependencies in the IDL widget toolkit. Consult the descriptions of the individual `DIALOG` and `WIDGET` routines in the *IDL Reference Guide* for complete details.

## Dialog Routines

IDL's `DIALOG_` routines (`DIALOG_PICKFILE`, etc.) rely on operating system native dialogs for most of their functionality. This means, for example, that when you use `DIALOG_PICKFILE` in an IDL application, a Windows user will see the Windows-native file selection dialog, a Macintosh user will see the appropriate Macintosh-native file selection dialog (there are two), and Motif users will see the Motif file selection dialog. Consult the descriptions of the individual `DIALOG` routines in the *IDL Reference Guide* for notes on the platform dependencies.

## Base Widgets

Base widgets (created with the `WIDGET_BASE` routine) play an especially important role in creating widget-based IDL applications because their behavior controls the way the application and its components are iconized, layered, and destroyed. See [Iconizing, Layering, and Destroying Groups of Top-Level Bases](#) under `WIDGET_BASE` in the *IDL Reference Guide* for details about the platform-dependent behavior.

## Positioning Widgets within a Base Widget

The widget geometry management keywords to the `WIDGET_BASE` routine allow a great deal of flexibility in positioning child widgets within a base widget. When building cross-platform applications, however, making use of IDL's explicit positioning features can be counterproductive.

Because IDL attempts to provide a platform-native look on each platform, widgets depend on the platform's current settings for font, font size, and "window dressing" (things like the thickness of borders and three-dimensional appearance of controls). As a result of the platform-specific appearance of each widget, attempting to position individual widgets manually within a base will seldom give satisfactory results on all platforms. Instead, insert widgets inside base widgets that have the `ROW` or

COLUMN keywords set, and let IDL determine the correct geometry for the current platform automatically. You can gain a finer degree of control over the layout by placing groups of widgets within sub-base widgets (that is, base widgets that are the children of other base widgets). This allows you to control the column or row layout of small groups of widgets within the larger base widget.

In particular, refrain from using the X/YSIZE and X/YOFFSET keywords in cross-platform applications. Using the COLUMN and ROW keywords instead will cause IDL to calculate the proper (platform-specific) size for the base widget based on the size and layout of the child widgets.

## Fonts used in Widget Applications

You can specify the font used in a widget via the FONT keyword. In general, the default fonts used by IDL widgets will most closely approximate the look of a platform-native application. If you choose to specify the fonts used in your widget application, however, note that the different platforms have different font-naming schemes for device fonts. While device fonts will provide the best performance for your application, specifying device fonts for your widgets requires that you write platform-dependent code as described in “Which Operating System is Running?” on page 625. You can avoid the need for platform-dependent code by using the TrueType fonts supplied with IDL; there may be a performance penalty when the fonts are initially rendered. See [Appendix G, “Fonts”](#) in the *IDL Reference Guide* for details.

## Application Menu Bars

The Macintosh is unique among the platforms on which IDL runs in that it provides a single menu bar at the top of the screen for the currently-active application. The APP\_MBAR keyword to the WIDGET\_BASE function allows your application to “take over” the Macintosh system menu when your IDL application is active. If you wish to place the menu for your application in an individual window, use the MBAR keyword instead. Code that uses the APP\_MBAR keyword acts as if the MBAR keyword had been specified. See [APP\\_MBAR](#) under WIDGET\_BASE in the *IDL Reference Guide* for details.

## Motif Resources

Use the RESOURCE\_NAME keyword to apply standard X Window System resources to a widget on a Motif system. Resources specified via the RESOURCE\_NAME keyword will be quietly ignored on Windows and Macintosh systems. See [RESOURCE\\_NAME](#) under WIDGET\_BASE in the *IDL Reference*



*Guide* for details. In general, you should not expect to be able to duplicate the level of control available via X Window System resources on other platforms.

## WIDGET\_STUB

On Motif platforms, you can use the `WIDGET_STUB` routine to include widgets created outside IDL (that is, with the Motif widget toolkit) in your IDL applications. The `WIDGET_STUB` mechanism is only available under Unix and VMS, and is thus not suitable for use in cross-platform applications that will run under Microsoft Windows or on the Macintosh. `WIDGET_STUB` is described in the *External Development Guide*.

## Widget Event Inconsistencies

Different windowing systems provide different types of events when graphical items are displayed and manipulated. IDL attempts to provide consistent functionality on all windowing systems, but is not always completely successful. Two inconsistencies that have caused confusion in the past are:

- Enter/Exit tracking events are not generated by some windowing systems. IDL attempts to provide appropriate enter/exit events, but behaviors may differ on different platforms.
- When an IDL draw widget (created with the `WIDGET_DRAW` function) is realized on screen, an expose event is generated under Microsoft Windows but not on other platforms.

Handle individual widget events carefully, and be sure to test your code on all platforms supported by your application.

## Using External Code

The use of programs written in languages other than IDL—either by calling code from an IDL program via `CALL_EXTERNAL` or `LINKIMAGE` or via the callable IDL mechanism—is an inherently platform-dependent process. Writing a cross-platform IDL program that uses `CALL_EXTERNAL` or `LINKIMAGE` requires that you provide the appropriate programs or shared libraries for each platform your application will support, and is beyond the scope of this chapter. Similarly, the Callable IDL mechanism is necessarily different from platform to platform. See the *External Development Guide* for details on writing and using external code along with IDL.

## IDL DataMiner Issues

The IDL DataMiner provides a platform-independent interface to IDL's Open Database Connectivity (ODBC) features. Note, however, that the ODBC drivers that allow connection to different databases are platform-dependent, and may require platform-dependent coding. In addition, the dialogs called by the `DIALOG_DBCONNECT` function are provided by the specific ODBC driver in use, and will be different from data source to data source.





Chapter 21:

# Extending the IDL Online Help System

The following topics are covered in this chapter:

---

Overview .....	646	Creating Hypertext Files for Use with IDL's Hypertext Help Viewer .....	647
----------------	-----	--	-----

## Overview

Users may want to create online help for their own IDL applications, procedures, or functions. The online help system used by IDL emulates (or uses, in the case of IDL for Windows) the Microsoft Windows Help viewer on all supported platforms. Because the online help files are *compiled*, there is not a simple, no-cost way to include user-created help topics directly in the help system on all platforms. However, there *are* a number of ways to create your own online help. The techniques described in this section vary in complexity, cost, and level of integration with IDL's hypertext online help viewer.

# Creating Hypertext Files for Use with IDL's Hypertext Help Viewer

The online help system used by IDL emulates (or uses, in the case of IDL for Windows) the Microsoft Windows Help viewer on all supported platforms. It is possible to create your own hypertext help files that can be used with the viewer. The difficulty and expense involved in creating such files depends largely on the platform(s) involved.

## Microsoft Windows

For Microsoft Windows systems, help files are relatively easy to create. Files must be created in the Rich Text Format (RTF) and compiled with Microsoft's help compiler. The help compiler is part of the Windows Software Developer's Kit, and is now included in several Microsoft programming products, including the Visual C++ development environment. The help compiler may also be available from the Microsoft ftp site (<ftp.microsoft.com>) or other Microsoft online software libraries at little or no cost.

The Windows help system is often referred to as "WinHelp". The two components are the viewer (`WINHELP.EXE`, found in the main `WINDOWS` directory of all Windows systems), and the help compiler. There are a number of third-party "help authoring systems" that simplify the creation of WinHelp compatible RTF files. Also, a number of third party books describe the WinHelp creation process—*Developing Online Help for Windows*, by Scott Boggan, David Farkas, and Joe Welinske, Sams Publishing, 1993, ISBN: 0-672-30230-6 is one that we have found useful.

## Macintosh

For Macintosh, we use a WinHelp-compatible compiler and viewer licensed from Altura Software, Inc. called QuickHelp. This compiler uses the same RTF files as are used by the Microsoft Help compiler. Altura Software can be contacted at the following address:

Altura Software, Inc.  
510 Lighthouse Avenue, Suite 5  
Pacific Grove, CA 93950  
Phone: 408-655-8005  
Fax: 408-655-9663  
AppleLink: ALTURA

## UNIX and VMS

For UNIX and VMS, we use a compiler and viewer from Bristol Technology, Inc. called HyperHelp. Bristol makes a number of compilers that can compile source files in RTF, FrameMaker's MIF (Maker Interchange Format), SGML (Standard Generalized Markup Language), HTML (Hypertext Markup Language), and their own simple HyperHelp Text (HHT) format. We use the MIF compiler to create hypertext help files from the same FrameMaker files that produce our hardcopy manuals.

Bristol also makes a product called Bridge that takes compiled HyperHelp files and converts them to RTF files that can be compiled with the Windows and Macintosh help compilers described above. In this way, we can create help files for all supported IDL platforms from a single source.

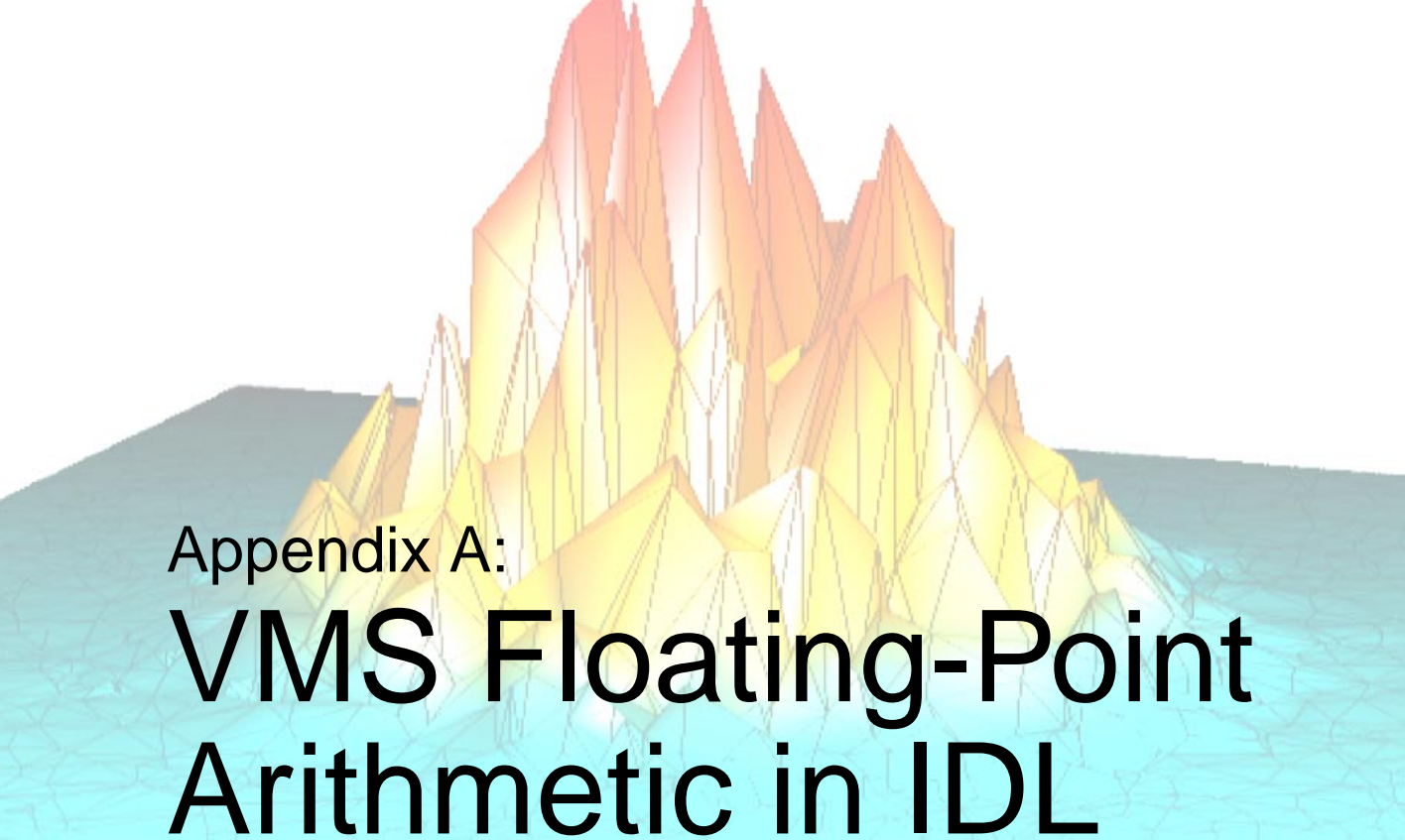
Bristol Technology can be contacted at the following address:

Bristol Technology, Inc.  
241 Ethan Allen Highway  
Ridgefield, CT 06877  
Phone: 203-438-6969  
Fax: 203-438-5013  
E-mail: [info@bristol.com](mailto:info@bristol.com)  
<http://www.bristol.com>

## Accessing Hypertext Help Files

Once compiled, your own hypertext help files can be opened by selecting "Open" from the help viewer's File menu. In addition, the `ONLINE_HELP` procedure can be used in your IDL programs to display help files and control the viewer. See [ONLINE\\_HELP](#) in the *IDL Reference Guide* for details.





Appendix A:

# VMS Floating-Point Arithmetic in IDL

The following topics are discussed in this appendix:

---

Overview .....	650	A Strategy for Converting VMS Programs	656
VAX Floating-Point Format Background .	651	Using CALL_EXTERNAL .....	658
Transition Issues .....	653	A Note on the VMS G Float Format .....	660
A Warning About Floating-Point Conversions in IDL .....	655		

# Overview

All VMS versions of IDL through release 5.0 used VAX F and D floating-point formats. In contrast, all non-VMS versions of IDL use a different and more modern floating-point standard (IEEE 754). Starting with IDL release 5.1, VMS IDL has been converted to also support IEEE floating-point formats rather than the now obsolete VAX formats. This appendix explains the history behind these decisions and discusses how to convert older VMS IDL programs.

# VAX Floating-Point Format Background

The floating-point format used by a program such as IDL is determined entirely by the computer hardware upon which it runs. In the early years of computing it was common for different machines to have incompatible floating-point formats. In the 1970s and 1980s, PDP-11 and VAX minicomputers were widely used for scientific computation, and their floating-point format (known as F and D floating) became the *de facto* standard for science.

Over the years, the computing industry has converged upon a floating-point standard known as IEEE 754, and commonly referred to as “ieee floating” or “ieee arithmetic”, and other formats (including the VAX) have diminished in importance. Now, all common computing hardware uses the IEEE formats, which has significant advantages over earlier ones:

- Binary data is portable to almost all current and foreseeable computing hardware and operating systems, requiring at most simple byte swapping.
- Special *Infinity* and *Not A Number (NaN)* values for undefined computations allow exceptional computations to be carried out in a well defined manner.

This convergence gained momentum in the 1980s as workstations and personal computers came into prominence. The result is that non-VMS versions of IDL have always used IEEE floating-point—a significant difference between them and the VMS version. VAX/VMS IDL stayed with the VAX formats to provide backwards compatibility with existing programs and data, and because the VAX hardware does not support the IEEE formats.

In the early 1990s, Digital Equipment Corporation released a new hardware architecture named ALPHA to replace the aging VAX line. The native floating-point format for ALPHA is IEEE, but it also supports the VAX formats for backwards compatibility. When IDL was ported to ALPHA/VMS, it was tempting to switch to IEEE floating-point in order to bring it in line with all other computers. However, the decision was made to stay with the VAX formats in order to maximize compatibility with our VMS customers existing binary data. Since support for the VAX was not discontinued at that time, Research Systems felt that it was important for all VMS implementations to be compatible with each other.

With IDL 5.1, the floating-point format for ALPHA/VMS IDL has been changed to IEEE. There are many reasons for this decision:

- IDL no longer runs on VAX hardware, and ALPHA supports IEEE natively. There is no longer a hardware barrier to conversion.

- The VAX formats are obsolete and no longer supported by any modern computing hardware, making an eventual switch inevitable.
- The lack of special *NaN* and *Infinity* values prevented important IDL features from being useful under VMS, and differences in floating-point precision made some numerical methods behave differently than on the other platforms.
- The ALPHA implementation of VAX D float has three fewer bits of precision than VAX hardware, making that format even less attractive.
- Unlike the past, it is rare for computing sites to be VMS-only these days. Most VMS users also use Unix workstations and personal computers, and those machines all use the IEEE floating-point representation. These sites have already addressed the issue of moving data between these formats, and are therefore in a position to move to IEEE under VMS. For most sites, the fact that VMS IDL did not use IEEE floating-point had become a primary barrier to moving completely beyond the transition to IEEE.

# Transition Issues

Most existing VMS applications will work with the IEEE version of VMS IDL without code changes. The slight differences in precision and range between the VAX and IEEE formats do not usually cause problems as long as you are aware of the limitations discussed in [“A Warning About Floating-Point Conversions in IDL”](#) on page 655. Since most VMS sites also use non-VMS computers, such conversions are probably already common at your site.

Transition issues therefore center around permanent data kept in disk files and off-line storage. Within IDL, the focus is therefore on data entering and leaving IDL:

## Input/Output

Programs that read binary data in VAX format will have to convert the data to IEEE format so that IDL can understand it. Similarly, programs that write data to a file that is supposed to contain VAX format data must convert the data to VAX format before writing it. The `BYTEORDER` procedure has a number of new keywords designed to perform this operation. More conveniently, the `VAX_FLOAT` keyword to the `OPEN` routines causes all binary data input or output via `ASSOC`, `READU`, or `WRITEU` to be automatically converted to the VAX floating-point format.

## CALL\_EXTERNAL

Programs that pass floating-point data to or from code dynamically linked to IDL using `CALL_EXTERNAL` may need to be adjusted. The options, in order of preference, are:

1. Recompile the linked code to use the IEEE floating-point format (that is, using the `/IEEE_FLOAT/IEEE_MODE=DENORM` compiler options).
2. Use the `VAX_FLOAT` keyword to `CALL_EXTERNAL` to automatically translate all data to and from VAX format as necessary.
3. Convert data to or from VAX format using `BYTEORDER`.

Again, remember that conversion from one format to another is not without consequences. Please read [“A Warning About Floating-Point Conversions in IDL”](#) on page 655 before making a final decision.

## LINKIMAGE

Programs that pass floating-point data to or from code dynamically linked to IDL using `LINKIMAGE` *require* that the linked code be recompiled to use the IEEE floating-point format (that is, using the `/IEEE_FLOAT/IEEE_MODE=DENORM` compiler options).

## SAVE and RESTORE

While data also enters and leaves IDL via the `SAVE` and `RESTORE` procedures, there is no IEEE transition issue for such data. The portable XDR format of `SAVE` files is already compatible with IEEE. Furthermore, `RESTORE` automatically converts the data in old VMS format `SAVE` files to IEEE format as it reads the data, allowing data in the older format to be recovered as well.

# A Warning About Floating-Point Conversions in IDL

The VAX formats are obsolete and IEEE is the standard for modern computing hardware. With or without IDL, you will eventually find it necessary to convert your existing VAX data to IEEE format if it is to remain usable. In doing so, you should understand that translation of floating-point values from one format to the other and back is not a completely reversible operation, and should be avoided when possible. Two important differences between the VAX and IEEE formats can lead to data loss:

1. The VAX floating-point format lacks support for the IEEE special floating-point values *NaN* and *Infinity*. Their special meaning is lost when they are converted to VAX format, and the meaning cannot be recovered.
2. Differences in precision and range can also cause information to be lost in both directions.

The conversion of existing VAX format data to IEEE cannot be avoided, and the information lost is usually small. Once the data is converted to IEEE, however, it is best to keep it and any results computed from it in IEEE format and avoid converting it back to the VAX format for storage.

For this reason, we recommend recompiling all code called via `CALL_EXTERNAL` to use the IEEE floating-point format rather than using the `VAX_FLOAT` keyword. New data should be written to files in IEEE format whenever possible.

# A Strategy for Converting VMS Programs

Starting with IDL 5.1, all IDL platforms, including ALPHA/VMS, use the IEEE floating-point format. VMS sites upgrading from a previous version of IDL are faced with the issue of how to manage this conversion. We recognize that such a conversion cannot occur all at once, and will instead be carried out gradually. We suggest the following general approach to making the transition.

## Step 1: Ensure the Stability of Existing Operations

To ensure that your applications continue to work, keep IDL 5.0.x installed on your systems, and use it to run existing applications. Install the IEEE version of IDL in parallel with the older 5.0.x version and keep both available. Then, shift to the newer IEEE version as applications and data are ported.

## Step 2: Use Compatibility Mode To Make The Initial Port

IEEE versions of VMS IDL can be started with the `/VAX_FLOAT` command qualifier. This causes the default value of the `VAX_FLOAT` keywords to `OPEN` and `CALL_EXTERNAL` to be `TRUE` instead of `FALSE` as is usually the case. This is often sufficient to allow programs that do not use `LINKIMAGE` to run with IDL 5.1 while preserving the VAX format of all external data.

### Note

---

You can also use the `VAX_FLOAT` function to check or change the default value of the keywords to `OPEN` and `CALL_EXTERNAL` at runtime.

---

## Step 3: Full Port

To move your code fully to IEEE VMS without using the special compatibility mode (the `/VAX_FLOAT` command qualifier or calls to the `VAX_FLOAT` function) you will need to take the following steps:

1. Recompile dynamically linked code to use the IEEE floating-point format (that is, using the `/IEEE_FLOAT/IEEE_MODE=DENORM` compiler options).
2. If possible, convert data files to IEEE format, using the IEEE version of IDL to read the VAX format data and then write a new version of the file in IEEE format.



3. If data files cannot be converted to IEEE format, adjust the OPEN statements that access them to include the VAX\_FLOAT keyword so that IDL converts the data on input and output.

## Using CALL\_EXTERNAL

The `VAX_FLOAT` keyword to `CALL_EXTERNAL` can be used to make the IEEE versions of VMS IDL properly pass floating-point data to external code compiled for the VAX floating-point format. However, IDL 5.0.x and earlier did not accept this keyword. This makes it difficult to write a `CALL_EXTERNAL` statement that can be used under both versions at the same time.

The `VAX_CALL_EXT` routine shown below can be used to solve this problem. If you compile and use `VAX_CALL_EXT` it instead of `CALL_EXTERNAL`, your program will be able to specify the `VAX_FLOAT` keyword in all cases, and the older IDL versions will simply ignore the keyword as a side effect of keyword inheritance.

Enter the following IDL code in a file named `call_ext.pro` and include it in your IDL path. Then use the `VAX_CALL_EXT` function with the `VAX_FLOAT` keyword wherever you would otherwise use `CALL_EXTERNAL`.

```

FUNCTION vax_call_ext, a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, $
    a11, a12, a13, a14, a15, a16, _EXTRA=e
CASE N_PARAMS() OF
    2: ans = CALL_EXTERNAL(a1,a2,_EXTRA=e)
    3: ans = CALL_EXTERNAL(a1,a2,a3,_EXTRA=e)
    4: ans = CALL_EXTERNAL(a1,a2,a3,a4,_EXTRA=e)
    5: ans = CALL_EXTERNAL(a1,a2,a3,a4,a5,_EXTRA=e)
    6: ans = CALL_EXTERNAL(a1,a2,a3,a4,a5,a6,_EXTRA=e)
    7: ans = CALL_EXTERNAL(a1,a2,a3,a4,a5,a6,a7,_EXTRA=e)
    8: ans = CALL_EXTERNAL(a1,a2,a3,a4,a5,a6,a7,a8,_EXTRA=e)
    9: ans = CALL_EXTERNAL(a1,a2,a3,a4,a5,a6,a7,a8,a9,_EXTRA=e)
   10: ans = CALL_EXTERNAL(a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,_EXTRA=e)
   11: ans = CALL_EXTERNAL(a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11, $
        _EXTRA=e)
   12: ans = CALL_EXTERNAL(a1,a2,a3,a4,a5,a6,a7,a8,a9,a10, $
        a11,a12,_EXTRA=e)
   13: ans = CALL_EXTERNAL(a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,$
        a11,a12,a13,_ex,tra=e)
   14: ans = CALL_EXTERNAL(a1,a2,a3,a4,a5,a6,a7,a8,a9,a10, $
        a11,a12,a13,a14,_EXTRA=e)
   15: ans = CALL_EXTERNAL(a1,a2,a3,a4,a5,a6,a7,a8,a9,a10, $
        a11,a12,a13,a14,a15,_EXTRA=e)
   16: ans = CALL_EXTERNAL(a1,a2,a3,a4,a5,a6,a7,a8,a9,a10, $
        a11,a12,a13,a14,a15,a16,_EXTRA=e)
ENDCASE
RETURN, ans
END

```

**Note**

---

You do not need to enter the code for `VAX_CALL_EXT.PRO` by hand. It is included in the `misc` subdirectory of the `examples` directory of the IDL distribution.

---

## A Note on the VMS G Float Format

In addition to the F and D floating-point formats, VMS systems also support a format known as G float, which has fewer mantissa bits than D float and larger range. On the VAX, this format was rarely used, and IDL has never supported it. Under ALPHA/VMS, however, G float is the default for DEC language compilers. This makes it very easy to inadvertently build programs that produce G float data. G float offers little advantage, if any, over double precision IEEE, while causing compatibility issues similar to those caused by F and D float. For this reason, Research Systems recommends that you not use G float unless you have specific requirements for it. To compile your programs to produce IEEE format floating point, specify the `/IEEE_FLOAT` command qualifier to the compiler. There are several levels of compiler support for IEEE math, controlled by the `/IEEE_MODE` qualifier. IDL is built with the options `/IEEE_FLOAT/IEEE_MODE=DENORM`.

If you need to use G float data with IDL, you will need to manually convert the data to and from IEEE format. The `DTOGFLOAT` and `GFLOATTOD` keywords to the `BYTEORDER` procedure can be used for this task.



# Index

## *Symbols*

!ERROR\_STATE system variable, [333](#), [343](#)  
  MSG, [340](#)  
  SYS\_MSG, [340](#)  
# of Rows/Columns base property, [496](#)  
# operator, [120](#)  
## operator, [120](#)  
.prc file  
  testing in project, [30](#)  
.prj files, [23](#)  
< operator, [119](#)  
<nopage>controls *see* widgets, [542](#)  
> operator, [119](#)  
?: ternary operator, [126](#)  
^ character, [119](#)  
\_EXTRA keyword (keyword inheritance), [291](#)

\_REF\_EXTRA keyword (keyword inheritance), [291](#)

## *Numerics*

64-bit data type  
  long, [102](#)  
  unsigned long, [102](#)

## *A*

abbreviating keywords, [286](#)  
about IDL, [15](#)  
active command line, [565](#)  
actual parameters, [286](#)  
adding  
  files to a project, [26](#)

- addition operator, 118
- Alignment, 497
- Alignment base property, 497
- Alignment button property, 508
- Alignment label property, 516
- Alignment table property, 532
- Allow Closing base property, 497
- Allow Moving base property, 498
- Altura Software, 647
- AND operator, 122
- animation
  - compound widget, 546
- anonymous structures, 93, 132
- applications, written in IDL, 14
- ARG\_PRESENT function, 307
- arithmetic errors, 345
- arrays
  - concatenation, 121
  - definition, 90
  - efficient accessing, 410
  - of structures, 143
  - subscripts
    - definition, 91
    - ranges, 161
- assignment, 269
  - pointers, 239
  - statement, 198
- assignment operator, 118
- ASSOC function, 354, 357
- associated I/O, 406
- attributes
  - draw widget, 526
  - droplist, 521
  - label widget, 516
  - listbox, 523
  - table widget, 532
- automatic compilation, 216
- automatic structure definition, 149, 259

## B

- background tasks
  - for widgets, 580
- base widgets
  - attributes, 496
  - bulletin board bases, 584
  - defined, 544
  - events, 505
  - using, 454
- BEGIN statement, 204
- Bell, ringing the terminal, 109
- binary trees, 253
- bitmap
  - files
    - standard file format I/O routines, 435
- Bitmap button property, 508
- Bitmap Editor, 465
  - opening, 508
  - tools, 466
- bitmaps
  - adding to buttons, 465
- blocks, 204
- BMP files
  - adding to button widgets, 465
  - displaying on buttons, 508
  - standard file format I/O routines, 435
  - supplied, 465
- Boolean operators, 122
- breakpoints
  - debugging with, 611
- Bristol Technology, 648
- bubble sort, 251
- bugs
  - debugging in IDL, 609
- building
  - order in project, 36
  - projects, 39
- bulletin board bases, 584
- button widgets
  - adding menus to, 464
  - defined, 544

- setting attributes, [508](#)
- setting properties, [507](#)
- using, [454](#)

byte

- arguments and strings, [177](#)
- data type, [102](#)

BYTE function, [178](#)

## C

CALL\_EXTERNAL, [653](#), [658](#)

CALL\_FUNCTION function, [312](#)

CALL\_PROCEDURE procedure, [312](#)

calling

- mechanism for procedures, [300](#)

CALLS keyword, [342](#)

CANCEL keyword, [335](#)

caret, [119](#)

carrot, [119](#)

case folding, [179](#)

CASE statement, [206](#)

CATCH procedure, [333](#)

change value event, [519](#)

changing widget values, [578](#)

characters

- non-printable, [109](#)

CHECK\_MATH function, [345](#)

checkbox widgets

- creating, [507](#)
- laying out, [508](#)
- setting attributes, [508](#)
- setting properties, [507](#)

checkboxes, [507](#)

- using, [455](#)

class

- object, [257](#)
- structure, [259](#)
- structures

  - zeroed, [259](#)

closing

- files, [359](#)

- projects, [25](#)

code

- IDL GUIBuilder generated, [448](#)
- modifying generated, [449](#)

Color Model

- draw area property, [526](#)

color tables

- example, [450](#)

colors

- manipulation compound widgets, [546](#)

Colors draw area property, [526](#)

COLUMN keyword, [584](#)

Column Labels table property, [532](#)

comments, [197](#)

common blocks, [208](#)

- widgets and, [592](#)

compiling

- a file from a project, [29](#)
- all files in a project, [38](#)
- modified files in a project, [38](#)

compiling functions and procedures, [302](#)

complex

- numbers, [85](#), [107](#)

complex data type, [103](#)

Component Sizing common property, [490](#)

compound widgets, [546](#), [594](#)

- animation, [546](#)
- color manipulation, [546](#)
- data entry, [546](#)
- example, [480](#)
- handling events, [493](#)
- image manipulation, [547](#)
- in IDL GUIBuilder code, [480](#)
- orientation, [547](#)
- user interface, [547](#)
- writing, [598](#)

concatenation

- array, [121](#)
- string, [175](#)

conditional expression, [126](#)

- constants
  - complex, 85, 107
  - decimal, 104
  - double-precision, 84, 106
  - floating-point, 84, 106
  - hexadecimal, 104
  - integer, 84, 104
  - ivalues, 84, 105
  - octal, 104
  - string, 86, 87, 107
- context, 332
- creating
  - .sav file from a project, 39
  - heap variables, 233
  - IDL runtime distribution, 43, 52
  - projects, 23
- creating multiple, 507
- cursor
  - hourglass, 579
- CW\_DICE function, 598
- CW\_PDMENU function, 575
  
- D**
- dangling references, 243, 264
- data entry
  - compound widgets, 546
- data types
  - 64-bit
    - long, 102
    - unsigned long, 102
  - byte, 102
  - complex, 103
  - double-precision complex, 103
  - double-precision floating-point, 102
  - floating-point, 102
  - integer, 102
  - long integer, 102
  - string, 103
  - unsigned
    - integer, 102
    - long, 102
- debugging, 609
- decimal, 104
- defining method routines, 273
  - note for Windows 3.11 users, 276
- deleting
  - files in a project, 28
- delimiters, string, 86, 108
- dereference operator, pointers, 239
- DESTROY keyword, 578
- destroying
  - objects, 268
  - widgets, 578
- developer's kit license, 44
- DIALOG\_PRINTERSETUP function, 548
- DIALOG\_PRINTJOB function, 548
- dialogs
  - file selection, 547
  - printing, 548
- disappearing variables, 332
- Display Headers table property, 532
- distributing IDL applications, 14
- distribution
  - creating, 43, 52
- division operator, 119
- DO statement, 204
- double-precision
  - complex data type, 103
  - floating-point data type, 102
- draw widgets, 571
  - attributes, 526
  - backing store, 528
  - color model, 526
  - colors used in, 526
  - defined, 544
  - events, 529
  - example application using, 445
  - graphics type, 527
  - mouse events, 529
  - mouse motion events, 530
  - properties, 526



- renderer type, 527
- scrolling, 528
- scrolling area, 528, 529
- using, 456
- view change events, 530
- viewport move, 531
- droplist widgets
  - defined, 545
  - events, 521
  - initial value, 521
  - select event, 522
  - setting attributes, 521
  - setting properties, 521
  - title, 521
  - using, 455
- DYNAMIC\_RESIZE keyword, 584

## E

- Editable table property, 533
- Editable text property, 511
- editing
  - a source file from a project, 29
- efficiency
  - constants, 320
  - if statements, 315
  - programming, 306
  - system functions and procedures, 319
  - vector and array operations, 317
- encapsulation, 257
- END statement, 204
- Entering Procedure Definitions, 296
- EOF function, 354
- EQ operator, 125
  - object references, 270
  - pointers, 242
- errors
  - default error-handling mechanism, 331
  - floating-point underflow, 345
  - handling, 330
    - CATCH procedure, 333
    - input/output, 338
    - ON\_ERROR procedure, 337
  - input/output, 338
  - math, 345
  - signaling (MESSAGE procedure), 340
  - system variables, 343
  - system variables for, 343
- event driven programming, 542
- events, 535
  - button press, 510
  - common properties, 493
  - compound, handling, 493
  - destruction, 494
  - draw area mouse, 529
  - draw area mouse motion, 530
  - draw area view changes, 530
  - draw area widget, 529
  - draw viewport move, 531
  - droplist, 521
  - droplist select, 522
  - focus, 505
  - handling in IDL GUIBuilder code, 449, 473, 474, 478, 483
  - interrupting the event loop, 591
  - keyboard focus, 590
  - kill request, 505
  - listbox, 524
  - listbox selection, 524
  - post creation, 495
  - realize, 494
  - release for buttons, 509
  - setting button, 510
  - slider, 519
  - slider change value, 519
  - table cell select, 536
  - table column width change, 536
  - table data invalid, 538
  - table focus, 537
  - table insert character, 537
  - table insert string, 538
  - table text delete, 537

- table text selection, 538
- text delete, 513
- text focus, 513
- text inserts, 514, 514
- text selection, 514
- text widget, 513
- timer, 494, 580
- tracking, 494
  - widget, 563
- EXECUTE function, 312
- EXIT procedure, 313
- explicitly formatted I/O, 356, 370
- exponentiation operator, 119
- exporting
  - projects, 42, 51
- expressions
  - efficiency of evaluation, 314
  - structure of, 127, 129
  - type of, 127
- EXTRA keyword (keyword inheritance), 291

## F

- false, definition of, 220
- file
  - adding to a project, 26
  - compiling from a project, 29
  - compiling in a project, 38
  - editing from a project, 29
  - moving in a project, 27
  - removing from a project, 28
  - setting properties for a project, 30
- file units, 360
- files
  - closing, 359
  - end-of-file, 416
  - file units, *see* file units
  - flushing file units, 414
  - formats
    - BMP, 435
    - GIF, 435
    - Interfile, 435
    - JPEG, 435
    - NRIF, 435
    - PICT, 435
    - PNG, 435
    - PPM, 435
    - SRF, 435
    - TIFF, 435
    - X11 Bitmap, 435
    - XWD, 436
- help and information, 411
- IDL GUIBuilder
  - generated, 448
  - generating code, 471
  - generating resource, 471
  - IDL code, 471
  - regeneration, 472
  - resource, 471
- indexed, 426
- input/output, 353
- locating, 411
- logical unit number, 360
- Macintosh-specific information, 433
- manipulation operations, 411
- modifying generated, 449
- multiple structures, 409
- opening, 358
- pointer position, 415
- record-oriented, 424
- selection dialogs, 547
- storing in a project, 22
- VMS-specific information, 422
- Windows-specific information, 432
- FILES keyword, 411
- FIND\_BY\_UNAME keyword, 474
- FINDFILE function, 354, 411
- FINITE function, 348
- Floating base property, 498
- floating point conversions, 655
- floating-point
  - data type, 102

- errors, 345
  - underflow errors, 345
- floating-point format, 651
- FLUSH procedure, 354
- focus events, 590
- FOR statement, 204, 211
- formal parameters, 286
- Format Codes, 375
- FORMAT keyword, 176, 177
- formatted I/O, 356
- FORWARD\_FUNCTION statement, 217
- Frame common property, 491
- free format I/O, 356, 365
- FREE\_LUN procedure, 354
- freeing pointers, 247
- FSTAT function, 354, 411
- function definition statement, 216
- functions, 296
  - defining, 216
  - definition statements, 285
  - forward definition, 217

## G

- GE operator, 125
- geometry of widgets, 583
- GET\_KBRD function, 354, 416
- GET\_LUN procedure, 354
- GET\_SCREEN\_SIZE keyword, 587
- GIF files
  - standard file format I/O routines, 435
- GOTO statement, 219
- Graphics Type draw area property, 527
- Grid Layout base property, 499
- group
  - moving files in a project, 27
- GT operator, 125
- GUIBuilder, *see* IDL GUIBuilder

## H

- Handle Events common event, 493
- heap variables, 231, 263
  - creating, 233
  - leakage, 244, 264
  - object, 231, 257, 263, 263
  - pointer, 235
  - saving and restoring, 234
- Height listbox property, 523
- Height text property, 511
- HELP procedure, 354
- hexadecimal, 104
- horizontal slider, *see* slider widgets
- hourglass cursor, 579
- HOURGLASS keyword, 579
- HTML, 648
- HyperHelp, 648
- HyperText Markup Language, 648

## I

- IDL
  - applications, distributing, 14
  - Code Profiler, 614
  - pointers, 236
  - runtime licensing, 14
  - statements, 196
- IDL GUIBuilder, 438
  - # of Rows/Columns property, 496
  - Alignment label property, 516
  - Alignment property, 497, 508
  - Alignment table property, 532
  - Allowing Closing property, 497
  - Allowing Moving property, 498
  - base widget attributes, 496
  - base widget events, 505
  - base widget properties, 496
  - base widgets, using, 454
  - Bitmap Editor, 465
  - Bitmap property, 508

- button attributes, 508
- button widgets, using, 454
- buttons, adding bitmaps, 465
- buttons, adding menus, 464
- checkbox attributes, 508
- checkbox widgets, using, 455
- checkboxes, creating, 507
- Color Model draw area property, 526
- color table example, 450
- Colors draw area property, 526
- Column Labels table property, 532
- common events, 493
- compiling and running example, 452
- Component Sizing property, 490
- copying or cutting widgets, 469
- creating draw area, example, 445
- creating multiple checkboxes, 507
- creating multiple radio buttons, 507
- defining menus, example, 442
- deleting widgets, 470
- Display Headers table property, 532
- draw area events, 529
- draw widget properties, 526
- draw widgets, using, 456
- droplist attributes, 521
- droplist events, 521
- droplist properties, 521
- droplists, using, 455
- Editable table property, 533
- Editable text property, 511
- event code, example, 483
- event code, handling example, 474
- event code, integrating interfaces, 478
- event code, understanding, 473
- example application, 442
- files, generating multiple times, 472
- files, IDL code, 471
- files, portable resource, 471
- Floating property, 498
- Frame property, 491
- generating code, 448, 471
- generating resource files, 471
- Graphics Type draw area property, 527
- Grid layout property, 499
- Handle Events common event, 493
- Height listbox property, 523
- Height text property, 511
- horizontal slider, using, 455
- Initial Value droplist property, 521
- Initial Value listbox property, 523
- Initial Value text property, 512
- integrating multiple interfaces, 478
- Label property, 509
- label widget attributes, 516
- label widget properties, 516
- label widgets, using, 455
- Layout property, 499
- listbox attributes, 523
- listbox events, 524
- listbox properties, 523
- listbox widgets, using, 455
- Maximum Value slider property, 518
- menus, editing, 461
- Minimize/Maximize property, 500
- Minimum Value slider property, 518
- Modal property, 500
- modifying code, example, 449
- moving widgets, 469
- Multiple listbox property, 523
- Name property, 490
- No Release property, 509
- OnButton draw area event, 529
- OnButton Press event property, 510
- OnCellSelect table event, 536
- OnChangeValue slider event, 519
- OnColWidth table event, 536
- OnDelete table event, 537
- OnDelete text event, 513
- OnDestroy event property, 494
- OnExpose draw area event, 530
- OnFocus event property, 505
- OnFocus table event, 537

- OnFocus text event, [513](#)
- OnInsertCh text event, [514](#)
- OnInsertChar table event, [537](#)
- OnInsertString table event, [538](#)
- OnInsertString text event, [514](#)
- OnInvalidData table event, [538](#)
- OnKillRequest event property, [505](#)
- OnMotion draw area event, [530](#)
- OnRealize event property, [494](#)
- OnSelectValue droplist event, [522](#)
- OnSelectValue listbox event, [524](#)
- OnSizeChange event property, [506](#)
- OnTextSelect table event, [538](#)
- OnTextSelect text event, [514](#)
- OnTimer event property, [494](#)
- OnTracking event property, [494](#)
- OnViewportMoved draw area event, [531](#)
- operating on widgets, [468](#)
- parent base, changing for widget, [469](#)
- pasting widgets, [469](#)
- Position slider property, [518](#)
- PostCreation event property, [495](#)
- Properties dialog, [457](#)
- radio button attributes, [508](#)
- radio button widgets, using, [455](#)
- radio buttons, creating, [507](#)
- redoing operations, [470](#)
- Renderer draw area property, [527](#)
- Resize Columns table property, [533](#)
- resizing widgets, [469](#)
- Retain draw area property, [528](#)
- Row Labels table property, [534](#)
- Row/Column Major table property, [534](#)
- Scroll draw area property, [528](#)
- Scroll property, [501](#)
- Scroll table property, [534](#)
- Scroll text property, [512](#)
- selecting widgets, [468](#)
- Sensitive property, [491](#)
- setting button events, [510](#)
- setting button properties, [507](#)
- setting text widget attributes, [511](#)
- setting text widget events, [513](#)
- slider events, [519](#)
- slider properties, [518](#)
- smooth example, [451](#)
- Space property, [501](#)
- starting, [440](#)
- Suppress Value slider property, [519](#)
- System Menu property, [502](#)
- table events, [535](#)
- table widget attributes, [532](#)
- table widget properties, [532](#)
- table widgets, using, [456](#)
- test mode, [447](#)
- Text label property, [516](#)
- text widgets properties, [511](#)
- text widgets, using, [455](#)
- Title Bar property, [502](#)
- Title droplist property, [521](#)
- Title property, [502](#)
- Title slider property, [519](#)
- toolbar, [454](#)
- tools, [453](#)
- Type property, [510](#)
- undoing operations, [470](#)
- vertical slider, using, [455](#)
- Viewport Columns table property, [535](#)
- Viewport Rows table property, [535](#)
- Visible property, [503](#)
- Widget Browser, [483](#)
- Widget Browser, using, [460](#)
- widgets, changing parent base of, [469](#)
- widgets, cutting, copying or pasting, [469](#)
- widgets, deleting, [470](#)
- widgets, moving, [469](#)
- widgets, resizing, [469](#)
- widgets, selecting, [468](#)
- Width listbox property, [524](#)
- Width text property, [512](#)
- Word Wrapping text property, [512](#)
- writing event-handling code, [449](#)

- X Offset property, [492](#)
  - X Pad property, [503](#)
  - X Scroll draw area property, [528](#)
  - X Scroll property, [504](#)
  - X Size property, [492](#)
  - Y Offset property, [492](#)
  - Y Pad property, [504](#)
  - Y Scroll draw area property, [529](#)
  - Y Scroll property, [504](#)
  - Y Size property, [493](#)
  - IDL object overview, [257](#)
  - IDL objects, [266](#)
  - IDL\_TREE example routine, [253](#)
  - IEEE floating point, [651](#)
  - IEEE standard, [82](#), [346](#), [348](#)
  - IF statement, [220](#)
    - avoiding, [315](#)
  - images
    - image manipulation compound widgets, [547](#)
  - implicit self argument, [274](#)
  - infinity, undefined result, [346](#)
  - information about objects, [271](#)
  - Informational Routines, [307](#)
  - inheritance, [261](#)
    - object, [258](#)
  - Initial Value droplist property, [521](#)
  - Initial Value listbox property, [523](#)
  - Initial Value text property, [512](#)
  - input/output
    - associated, [406](#)
    - error handling, [338](#)
    - explicit format, [356](#), [370](#)
    - format codes, [375](#)
    - format reversion, [374](#)
    - formatted, [356](#)
    - free format, [356](#), [365](#)
    - magnetic tape, [429](#)
    - portable, [401](#)
    - unformatted, [355](#), [394](#)
      - portable, [401](#)
      - string variables, [394](#)
    - UNIX FORTRAN unformatted data files, [410](#)
    - XDR, [401](#)
  - instance
    - object, [257](#)
  - integer
    - constants, [84](#), [105](#)
    - conversions, errors in, [348](#)
    - data type, [102](#)
  - Interfile files
    - standard file format I/O routines, [435](#)
- ## J
- joining strings, [186](#)
  - JPEG files
    - standard file format I/O routines, [435](#)
- ## K
- keyboard
    - focus events, [590](#)
  - KEYWORD\_SET function, [307](#)
  - keywords
    - inheritance, [291](#)
    - parameters, [286](#)
    - passing, [289](#)
    - setting, [286](#)
  - killing widgets, [578](#)
- ## L
- Label button property, [509](#)
  - label widgets
    - attributes, setting, [516](#)
    - defined, [545](#)
    - setting properties, [516](#)
    - using, [455](#)
  - Layout base property, [499](#)
  - LE operator, [125](#)

- license
    - developer's kit, 44
  - lifecycle
    - methods, 266
    - routines, 266
  - linked lists, 248
    - using pointers to create, 248
  - LINKIMAGE, 654
  - list widgets
    - defined, 545
  - listbox widgets
    - attributes, 523
    - events, 524
    - initial value, 523
    - multiple selections, allowing, 523
    - selection events, 524
    - setting height, 523
    - setting properties, 523
    - using, 455
    - width, 524
  - location of widgets, 584
  - logical unit numbers, 360
  - long integer data type, 102
  - LT operator, 125
  - LUNs (logical unit numbers), 360
- M**
- magnetic tape, 429
  - main menu bar enhancements, 611
  - Maker Interchange Format, 648
  - managing the state of a widget application, 592
  - math errors, 345
  - mathematical operators, 118
  - matrices, multiplying, 120
  - maximum operator, 119
  - Maximum Value slider property, 518
  - Menu Editor, using, 461
  - menus, 573
    - editing in IDL GUIBuilder, 461
    - pulldown, 575
    - system, using, 502
  - MESSAGE procedure, 340
  - message widgets
    - defined, 548
  - method overriding, 277
  - methods, 273
    - defining routines, 273
    - Windows 3.11, 276
    - invocation, 270
    - object, 257
  - MIF, 648
  - Minimize/Maximize base property, 500
  - minimum operator, 119
  - Minimum Value slider property, 518
  - Modal base property, 500
  - modal dialogs, creating, 500
  - modulo operator, 119
  - moving
    - files in a project, 27
  - Multiple listbox property, 523
  - multiplication operator, 118
- N**
- N\_ELEMENTS function, 287, 308
  - N\_PARAMS function, 287, 309
  - Name common property, 490
  - named
    - structures, 93, 132
  - names
    - of variables, 97
  - NaN (not-a-number), 346
  - NaN values, 651
  - NE operator, 125
    - object references, 270
    - pointers, 242
  - negation operator, 118
  - No Release button property, 509
  - non-printable characters, 109
  - NOT operator, 122

## NRIF

standard file format I/O routines, 435

## O

OBJ\_CLASS function, 271

OBJ\_DESTROY function, 268

OBJ\_ISA function, 271

OBJ\_NEW function, 266

OBJ\_VALID function, 272

OBJARR function, 267

## object

class, 257

class structures, 259

encapsulation, 257

heap variables, 257

inheritance, 258, 261

instances, 257

lifecycle, 266

method routines, 273

persistence, 258

polymorphism, 257

object heap variables, 263

object oriented programming, 256

## objects

destroying, 268

heap variables, 231, 263

references for heap variables, 231

Obtaining Traceback Information, 342

octal, 104

ON\_ERROR procedure, 330, 337

OnButton draw area event, 529

OnButton Press event property, 510

OnCellSelect table event, 536

OnChangeValue slider event, 519

OnColWidth table event, 536

OnDelete table event, 537

OnDelete text event, 513

OnDestroy property, 494

OnExpose draw area event, 530

OnFocus event property, 505

OnFocus table event, 537

OnFocus text event, 513

OnInsertCh text event, 514

OnInsertChar table event, 537

OnInsertString table event, 538

OnInsertString text event, 514

OnInvalidData table event, 538

OnKillRequest event property, 505

## online help

extending, 646

ONLINE\_HELP procedure, 648

OnMotion draw area event, 530

OnRealize event property, 494

OnSelectValue droplist event, 522

OnSelectValue listbox event, 524

OnSizeChange event property, 506

OnTextSelect table event, 538

OnTextSelect text event, 514

OnTimer event property, 494

OnTracking event property, 494

OnViewportMoved draw area event, 531

## opening

projects, 25

opening files, 358

OpenVMS *see* VMS

operations on objects, 269

operations on pointers, 239

## operators, 117

addition, 118

AND, 122

array concatenation, 121

assignment, 118

Boolean, 122

division, 119

EQ, 125

exponentiation, 119

GE, 125

GT, 125

LE, 125

LT, 125

mathematical, 118



- matrix multiplication, 120
- maximum, 119
- minimum, 119
- modulo, 119
- multiplication, 118
- NE, 125
- NOT, 122
- OR, 122
- parentheses, 117
- precedence, 115
- relational, 124
- square brackets, 117
- subtraction and negation, 118
- XOR, 122
- options
  - setting for project, 33
- OR operator, 122
- orientation, 3-dimensional, 547
- overflow, integer, 349
  
- P**
- parameters
  - actual, 286
  - copying, 287
  - formal, 286
  - passing mechanism, 286, 298
- parentheses, 117
- passing parameters, 298
- performance
  - analyzing, 614
- persistence, 258
- PICT files
  - standard file format I/O routines, 435
- PNG files
  - standard file format I/O routines, 435
- POINT\_LUN procedure, 354
- pointer heap variables, 263
- pointers, 231, 263
  - examples, 248
  - examples of using, 248
  - freeing, 247
  - heap variables, 231, 235
  - validity, 246, 246
- polymorphism
  - objects, 257
- portable unformatted I/O, 401
- Position slider property, 518
- positional parameters, 286
- PostCreation event property, 495
- PPM files
  - standard file format I/O routines, 435
- prc file
  - testing in project, 30
- PRINT procedure, 354
- printing
  - dialog, 548
  - properties, 548
  - setup dialog, 548
- printing dialogs, 548
- PRINTNAMES example routine, 250
- prj files, 23
- procedures
  - call statement, 222
  - calling
    - mechanism, 300
  - definition statements, 285
- profiling, 614
- Program Control Routines, 312
- programming
  - routines, 306
- project
  - adding files, 26
  - closing, 25
  - compiling a file, 29
  - creating, 23
  - editing source files, 29
  - moving files, 27
  - opening, 25
  - removing files, 28
  - saving, 25
  - storing source files, 22

- testing a .prc file, 30
- projects
  - building, 39
  - compiling all files, 38
  - compiling modified files, 38
  - creating a .sav file, 39
  - exporting, 42, 51
  - overview, 20
  - running an application, 41
  - setting build order, 36
  - setting file properties, 30
  - setting options, 33
- properties
  - draw area widget, 526
  - entering multiple strings, 459
  - label widget, 516
  - table widget, 532
  - text widget, 511
- Properties dialog, 457
  - # of Rows/Columns base property, 496
  - Alignment base property, 497
  - Alignment button property, 508
  - Alignment label property, 516
  - Alignment table property, 532
  - Allow Moving base property, 498
  - Allowing Closing base property, 497
  - Bitmap button property, 508
  - Color Model draw area property, 526
  - Colors draw area property, 526
  - Column Labels table property, 532
  - Component Sizing common property, 490
  - Display Headers table property, 532
  - draw area events, 529
  - draw area widget properties, 526
  - droplist events, 521
  - droplist widgets, 521
  - Editable table property, 533
  - Editable text property, 511
  - entering multiple strings, 459
  - Floating base property, 498
  - Frame common property, 491
  - Graphics Type draw area property, 527
  - Grid Layout base property, 499
  - Handle Events common event, 493
  - Height listbox property, 523
  - Height text property, 511
  - Initial Value droplist property, 521
  - Initial Value listbox property, 523
  - Initial Value text property, 512
  - Label button property, 509
  - Layout base property, 499
  - listbox events, 524
  - listbox properties, 523
  - Maximum Value slider property, 518
  - Minimize/Maximize base property, 500
  - Minimum Value slider property, 518
  - Modal base property, 500
  - Multiple listbox property, 523
  - Name common property, 490
  - No Release button property, 509
  - OnButton draw area event, 529
  - OnButtonPress button event, 510
  - OnCellSelect table event, 536
  - OnChangeValue slider event, 519
  - OnColWidth table event, 536
  - OnDelete table event, 537
  - OnDelete text event, 513
  - OnDestroy common event, 494
  - OnExpose draw area event, 530
  - OnFocus base event, 505
  - OnFocus table event, 537
  - OnFocus text event, 513
  - OnInsertCh text event, 514
  - OnInsertChar table event, 537
  - OnInsertString table event, 538
  - OnInsertString text event, 514
  - OnInvalidData table event, 538
  - OnKillRequest base event, 505
  - OnMotion draw area event, 530
  - OnRealize common event, 494
  - OnSelectValue droplist event, 522
  - OnSelectValue listbox event, 524

OnSizeChange base event, [506](#)  
 OnTextSelect table event, [538](#)  
 OnTextSelect text event, [514](#)  
 OnTimer common event, [494](#)  
 OnTracking common event, [494](#)  
 OnViewportMoved draw area event, [531](#)  
 opening, [457](#)  
 Position slider property, [518](#)  
 PostCreation common event, [495](#)  
 Renderer draw area property, [527](#)  
 Resize Columns table property, [533](#)  
 Retain draw area property, [528](#)  
 Row Labels table property, [534](#)  
 Row/Column Major table property, [534](#)  
 Scroll base property, [501](#)  
 Scroll draw area property, [528](#)  
 Scroll table property, [534](#)  
 Scroll text property, [512](#)  
 Sensitive common property, [491](#)  
 setting label widget properties, [516](#)  
 Space base property, [501](#)  
 Suppress Value slider property, [519](#)  
 System Menu base property, [502](#)  
 table events, [535](#)  
 table widget properties, [532](#)  
 Text label property, [516](#)  
 Title Bar base property, [502](#)  
 Title base property, [502](#)  
 Title droplist property, [521](#)  
 Title slider property, [519](#)  
 Type button property, [510](#)  
 Viewport Columns table property, [535](#)  
 Viewport Rows table property, [535](#)  
 Visible base property, [503](#)  
 Width listbox property, [524](#)  
 Width text property, [512](#)  
 Word Wrapping text property, [512](#)  
 X Offset common property, [492](#)  
 X Pad base property, [503](#)  
 X Scroll base property, [504](#)  
 X Scroll draw area property, [528](#)

X Size common property, [492](#)  
 Y Offset common property, [492](#)  
 Y Pad base property, [504](#)  
 Y Scroll base property, [504](#)  
 Y Scroll draw area property, [529](#)  
 Y Size common property, [493](#)

## Q

QuickHelp, [647](#)  
 quotas, [326](#)  
 quotation marks, [86](#), [108](#)

## R

radio button widgets  
     creating, [507](#)  
     creating multiple, [507](#)  
     laying out, [508](#)  
     setting attributes, [508](#)  
     setting properties, [507](#)  
     using, [455](#)  
 READ procedure, [354](#), [355](#)  
 READNAMES example routine, [248](#)  
 READS procedure, [418](#)  
 realizing widgets, [578](#)  
 recommendations  
     storing files in a project, [22](#)  
 record-oriented files, [424](#)  
 recursion, [300](#)  
 REF\_EXTRA keyword (keyword inheritance), [291](#)  
 reference, parameters passed by, [298](#)  
 relational operators, [124](#)  
 relaxed structure assignment, [151](#)  
 REMOVE\_ALL keyword, [180](#)  
 removing  
     files in a project, [28](#)  
 Renderer draw area property, [527](#)  
 repeat statement, [226](#)

- Resize Columns table property, 533
- RESTORE procedure, 654
- restoring structures, 152
- Retain draw area property, 528
- Rich Text Format, 647
- ROW keyword, 584
- Row Labels table property, 534
- Row/Column Major table property, 534
- RTF, 647
- running
  - applications from a project, 41
- runtime IDL, 14

## S

- SAVE procedure, 654
- save/restore
  - heap variables, 234
- saving
  - projects, 25
- saving and restoring heap variables, 234
- SCR\_XSIZE keyword, 584
- SCR\_YSIZE keyword, 584
- screen size, finding, 587
- Scroll base property, 501
- Scroll draw area property, 528
- Scroll table property, 534
- Scroll text property, 512
- self argument (objects), 274
- semicolon, 197
- Sensitive common property, 491
- sensitizing widgets, 579
- setting
  - keywords, 286
  - options for a project, 33
  - properties of a file in a project, 30
- SGML, 648
- SINKSORT example routine, 251
- size
  - of widgets, 584
- SIZE function, 310
- sizing widgets, 583
- slider widgets, 519
  - defined, 545
  - displayed values, 519
  - initial position, 518
  - maximum value, 518
  - minimum value, 518
  - properties, 518
  - setting attributes, 518
  - setting events, 519
  - title, 519
  - using, 455
- smoothing
  - example, 451
- sorting
  - SINKSORT example, 251
- Space base property, 501
- spaces, removing from a string, 180
- splitting strings, 186
- square brackets, 117
  - See* arrays, concatenation
- SRF files
  - standard file format I/O routines, 435
- standard
  - image file formats, 435
- Standard Generalized Markup Language, SGML, 648
- statement labels, 197
- statements, 196
- stepping through, debugging, 610
- STOP procedure, 313
- storing
  - file in a project, 22
- STRCOMPRESS function, 180, 181
- string data type, 103
- STRING function, 176, 177, 417
- strings, 86, 87, 107
  - case folding, 179
  - concatenation, 175
  - finding last occurrence of substring in, 184
  - formatting data, 176

- length of, 182
  - nonstring arguments to routines, 174
  - operations, 173
  - substrings, 183
  - whitespace, 180
  - STRJOIN function, 186
  - STRLEN function, 182
  - STRLOWCASE function, 179
  - STRMATCH function, 188
  - STRMID function, 183, 185
  - STRPOS function, 183
  - STRPUT procedure, 183, 184
  - STRSPLIT function, 186
  - STRTRIM function, 180
  - STRUCT\_ASSIGN procedure, 151
  - structure of subarrays, 163
  - structures
    - advanced, 147
    - arrays of, 143
    - automatic definition, 149, 259
    - creating and defining, 93, 133, 149
    - definition, 151
    - input/output, 145
    - introduction to, 93, 132
    - number of fields in, 147
    - parameter passing, 95, 140
    - references, 94, 136
    - relaxed definition, 151
    - restoring, 152
    - using help with, 94, 139
    - zeroed, 133, 259
  - STRUPCASE function, 179
  - subscripts, 156
    - array valued, 165
    - arrays, 91
    - examples, 158
    - of scalars, 159
    - ranges, 161, 161
    - ranges, combined with arrays, 167
    - subscript arrays, 200, 202
  - substrings, 183
  - subtraction operator, 118
  - Suppress Value slider property, 519
  - suspending execution, 611
  - system
    - files, 326
    - System Menu base property, 502
    - system variables, 99
      - !ERROR\_STATE, 343
      - for errors, 343
- ## T
- table widgets, 535
    - alignment of text, 532
    - attributes, 532
    - cell select events, 536
    - column labels, 532
    - column width change events, 536
    - data invalid events, 538
    - data transfer to, 534
    - defined, 545
    - editing cells, 533
    - events, 535
    - focus events, 537
    - heading display, 532
    - height, 533
    - insert character events, 537
    - insert string events, 538
    - row labels, 534
    - scroll height, 535
    - scroll width, 535
    - scrolling, 534
    - sizing columns, 533
    - text delete events, 537
    - text selection events, 538
    - using, 456
    - width, 533
  - tabs, removing from a string, 180
  - TEMPORARY function, 325
  - ternary operator, ?:, 126
  - test mode, IDL GUIBuilder, 447

- testing
    - .prc file from a project, 30
  - Text label property, 516
  - text widgets
    - defined, 546
    - delete events, 513
    - focus events, 513
    - properties, 511
    - selection events, 514
    - setting attributes, 511
    - setting editable state, 511
    - setting height, 511
    - setting initial displays, 512
    - setting scrolling, 512
    - setting width, 512
    - setting word wrapping, 512
    - string insert events, 514
    - text insert events, 514
    - using, 455
  - TIFF files
    - standard file format I/O routines, 435
  - timer events (for widgets), 580
  - TIMER keyword, 580
  - Title Bar base property, 502
  - Title base property, 502
  - Title droplist property, 521
  - Title slider property, 519
  - toolbars
    - IDL GUIBuilder, 454
  - traceback information, 342
  - TREE\_EXAMPLE example routine, 253
  - trees, 248
    - binary, 253
  - true, definition of, 220
  - Type button property, 510
- U**
- underflow errors, 345
  - unformatted I/O, 355, 394
  - UNIX, OS-specific file I/O information, 419
  - unsigned data type
    - integer, 102
    - long, 102
  - UPDATE keyword, 587
  - user interface compound widgets, 547
  - user values
    - for widgets, 562
- V**
- value
    - parameters passed by, 298
    - widgets, 559
  - Variable Watch Window, 619
  - variables, 96
    - attributes of, 96
    - disappearing, 332
    - displaying current, 619
    - names of, 97
    - system, 99
  - VAX\_CALL\_EXT routine, 658
  - VAX\_FLOAT keyword, 658
  - vectors
    - subscripting, 161
  - vertical slider, *see* slider widgets
  - Viewport Columns table property, 535
  - Viewport Rows table property, 535
  - virtual memory, 306, 322
    - minimizing, 324
    - running out of, 323
    - system parameters, 325
  - Visible base property, 503
  - Visible property, 503
  - VMS
    - Open VMS
      - virtual memory performance
    - VMS file I/O information, 422

## W

- WAIT procedure, [313](#)
- while statement, [227](#)
- whitespace, removing from strings, [180](#)
- Widget Browser, [460](#), [483](#)
- WIDGET\_BASE function, [584](#)
- WIDGET\_CONTROL procedure, [578](#), [579](#)
- widgets, [542](#)
  - 3D orientation, [547](#)
  - application
    - errors, [556](#)
    - tips, [596](#)
  - attributes
    - slider, [518](#)
  - base, [544](#)
  - base focus events, [505](#)
  - base, alignment, [497](#)
  - base, allow moving, [498](#)
  - base, allowing closing, [497](#)
  - base, displaying titlebars, [502](#)
  - base, floating, [498](#)
  - base, grid layouts, [499](#)
  - base, kill request events, [505](#)
  - base, layouts, [499](#)
  - base, menus, using system, [502](#)
  - base, modal, [500](#)
  - base, resizing, [500](#)
  - base, rows and columns, [496](#)
  - base, scroll area size, [504](#)
  - base, scrolling, [501](#)
  - base, setting attributes, [496](#)
  - base, setting events, [505](#)
  - base, setting properties, [496](#)
  - base, spacing of contained widgets, [501](#)
  - base, spacing of widgets in, [503](#), [504](#)
  - base, titles, [502](#)
  - base, visibility, [503](#)
  - bases, using, [454](#)
  - Browser, [460](#)
  - button, press events, [510](#)
  - button, release events, [509](#)
  - button, setting properties, [507](#)
  - buttons, [544](#)
  - buttons, adding menus, [464](#)
  - buttons, displaying bitmaps, [508](#)
  - buttons, labels, [509](#)
  - buttons, using, [454](#)
  - changing values, [578](#)
  - checkboxes, using, [455](#)
  - common blocks and, [592](#)
  - common events, [493](#)
  - compound, [547](#), [594](#), [598](#)
  - compound, adding, [480](#)
  - compound, example, [480](#)
  - compound, handling events for, [493](#)
  - controlling, [578](#)
  - controlling visibility, [483](#)
  - creating in IDL GUIBuilder, [454](#)
  - creating with IDL GUIBuilder, [438](#)
  - destroy events, [494](#)
  - displaying, [483](#)
  - draw, [544](#), [571](#), [571](#)
  - draw area properties, [526](#)
  - draw area, color model, [526](#)
  - draw events, [529](#)
  - draw, attributes, [526](#)
  - draw, backing store, [528](#)
  - draw, changing view events, [530](#)
  - draw, colors used in, [526](#)
  - draw, graphic type, [527](#)
  - draw, mouse events, [529](#)
  - draw, mouse motion events, [530](#)
  - draw, render type, [527](#)
  - draw, scrolling, [528](#)
  - draw, scrolling area, [528](#), [529](#)
  - draw, using, [456](#)
  - draw, viewport move events, [531](#)
  - droplist, [545](#)
  - droplist attributes, [521](#)
  - droplist events, [521](#)
  - droplist properties, [521](#)
  - droplist, select events, [522](#)

- droplist, title, [521](#)
- droplists, initial value, [521](#)
- droplists, using, [455](#)
- dynamic resizing, [584](#)
- enabled or disabled state, [491](#)
- events, [563](#)
- examples, [550](#)
- explicit size, [583](#)
- finding screen size, [587](#)
- frames, using, [491](#)
- geometry, [583](#)
- height, [493](#)
- hierarchies, [578](#)
- hourglass cursor, [579](#)
- interrupting the event loop, [591](#)
- killing hierarchies, [578](#)
- label, [545](#)
- label, setting properties, [516](#)
- labels, using, [455](#)
- lifecycle, [554](#)
- list, [545](#)
- listbox attributes, [523](#)
- listbox events, [524](#)
- listbox properties, [523](#)
- listbox, height, [523](#)
- listbox, initial value, [523](#)
- listbox, multiple selections, [523](#)
- listbox, selection events, [524](#)
- listbox, using, [455](#)
- listbox, width, [524](#)
- location, [584](#)
- managing the state of applications, [592](#)
- menus, [573](#)
- message, [548](#)
- modal dialogs, [500](#)
- naming, [490](#)
- natural size, [583](#)
- portability, [596](#)
- positioning, [492](#)
- post creation events, [495](#)
- preventing layout flicker, [587](#)
- properties for IDL GUIBuilder, [457](#)
- pull-down menus, [575](#)
- radio buttons, using, [455](#)
- realize events, [494](#)
- realizing
  - hierarchies, [578](#)
- restarting after an error, [556](#)
- retrieving values, [578](#)
- sensitizing, [579](#)
- setting button events, [510](#)
- setting label attributes, [516](#)
- size, [584](#)
  - dynamic resizing, [584](#)
  - explicit, [583](#)
  - natural, [583](#)
  - sizing, [583](#)
- sizing, default or explicit, [490](#)
- slider, [545](#)
- slider properties, [518](#)
- slider, change value events, [519](#)
- slider, displayed values, [519](#)
- slider, initial position, [518](#)
- slider, maximum value, [518](#)
- slider, minimum value, [518](#)
- slider, setting events, [519](#)
- slider, title, [519](#)
- slider, using, [455](#)
- table, [545](#)
- table attributes, [532](#)
- table events, [535](#)
- table properties, [532](#)
- table, alignment, [532](#)
- table, cell select events, [536](#)
- table, column labels, [532](#)
- table, column width events, [536](#)
- table, data transfer to, [534](#)
- table, editing, [533](#)
- table, focus events, [537](#)
- table, heading display, [532](#)
- table, height, [533](#)
- table, insert character events, [537](#)



- table, insert string events, [538](#)
- table, invalid data events, [538](#)
- table, row labels, [534](#)
- table, scroll height, [535](#)
- table, scroll width, [535](#)
- table, scrolling, [534](#)
- table, sizing columns, [533](#)
- table, text delete events, [537](#)
- table, text selection events, [538](#)
- table, using, [456](#)
- table, width, [533](#)
- text, [546](#)
- text, character inserts, [514](#)
- text, delete event, [513](#)
- text, editable, [511](#)
- text, events, [513](#)
- text, focus events, [513](#)
- text, height, [511](#)
- text, initial display, [512](#)
- text, scrolling, [512](#)
- text, selection events, [514](#)
- text, string inserts, [514](#)
- text, using, [455](#)
- text, width, [512](#)
- text, word wrapping, [512](#)
- timer events, [494](#), [580](#)
- tracking events, [494](#)
- types, [544](#)
- user values, [562](#), [562](#)
- values, [559](#)
  - user, [562](#)
- width, [492](#)
- Width listbox property, [524](#)
- Width text property, [512](#)
- wildcards, in string searches, [188](#)
- windows
  - finding screen size, [587](#)
- WinHelp, [647](#)
- Word Wrapping text property, [512](#)
- wrapper routines, [291](#)

- WRITEU procedure, [354](#)
- writing
  - a compound widget, [598](#)

## X

- X Offset common property, [492](#)
- X Pad base property, [503](#)
- X Scroll base property, [504](#)
- X Scroll draw area property, [528](#)
- X Size common property, [492](#)
- X11 Bitmap, standard file format I/O routines, [435](#)
- XDICE procedure, [604](#)
- XDR, [401](#)
- XDR files, [357](#)
- XMANAGER procedure, [564](#), [567](#), [596](#)
- XOFFSET keyword, [584](#)
- XOR operator, [122](#)
- XREGISTERED function, [567](#)
- XSIZE keyword, [584](#)
- xwd files
  - standard file format I/O routines, [436](#)

## Y

- Y Offset common property, [492](#)
- Y Pad base property, [504](#)
- Y Scroll base property, [504](#)
- Y Scroll draw area property, [529](#)
- Y Size common property, [493](#)
- YOFFSET keyword, [584](#)
- YSIZE keyword, [584](#)

## Z

- zeroed structures, [133](#), [259](#)

