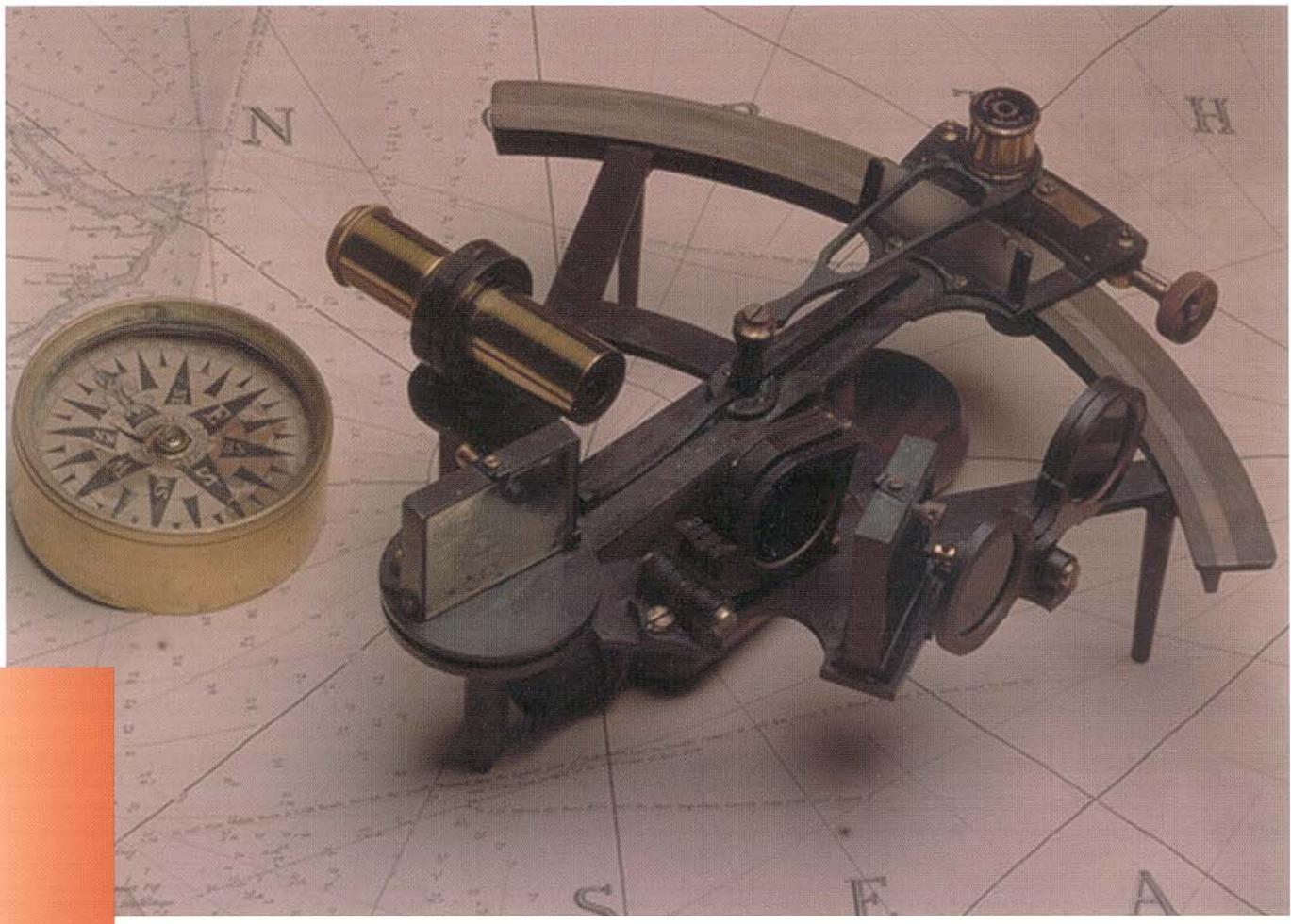


IDL Programming Techniques

Second Edition



David W. Fanning, Ph.D.

Created for Frederick Walter

IDL Programming Techniques

Second Edition

David W. Fanning, Ph.D.

Copyright © 2000 Fanning Software Consulting. All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Printed in the United States of America.

Published by Fanning Software Consulting
1645 Sheely Drive
Fort Collins, CO 80526
Phone: 970-221-0438
Fax: 970-221-0438
E-Mail: books@dfanning.com
Internet Address: <http://www.dfanning.com/>

Printing History

October 2000: First Printing

November 2000: Added information on double buffering of graphics displays.

IDL is a registered trademark of Research Systems, Inc.

Macintosh is a registered trademark of Apple Computer, Inc.

The X Window System is a trademark of the Massachusetts Institute of Technology.

Windows and Windows NT are trademarks of Microsoft Corporation.

UNIX is a registered trademark of UNIX System Laboratories.

ISBN 0-9662383-2-X



9 0000

9 780966 238327

ISBN 0-9662383-2-X
November 2000

Contents

♦ Discovering the Possibilities ♦♦♦

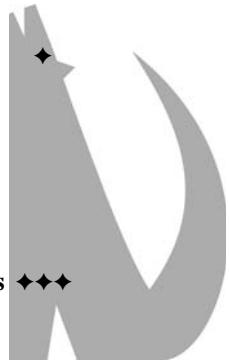


Table of Contents

Table of Contents	iii
Preface	xvii
Preface to the First Edition	xvii
Preface to the Second Edition	xviii
Getting Started	1
Chapter Overview	1
Philosophy Behind this Book	1
Using this Book	2
Required Version of IDL	2
Working with Colors in the IDL Session	2
Style Conventions Used in the Book	2
Capitalization	3
Comments	3
Line Continuation Characters	4
IDL Programs and Data Files Used in the Book	4
Installing the Program Files	4
Determining Your IDL Home and Current Directory	4
Downloading the Program Files Used with the Book	5
Make Sure Your Coyote Directory is on the IDL Path	5
Copying the Data Files	5
Obtaining Additional Help	6
Working with IDL Commands	6
Anatomy of an IDL Command	6
Positional Parameters	6
Keyword Parameters	7
IDL Procedures and Functions	7

Table of Contents

Help with IDL Commands	8
Creating Command Journals	8
Creating Variables	9
Variable Attributes Change Dynamically	10
Be Careful With Integer Variables	11
Working with Vectors and Arrays	13
Creating Vectors	13
Using Array Subscripts	13
Creating Arrays	14
Accessing Elements in Arrays	14
Extracting Vectors and Subarrays	15
Working with IDL Graphics Windows	15
Creating Graphics Windows	15
Determining the Current Graphics Window	16
Making a Graphics Window the Current Graphics Window	16
Deleting Graphics Windows	17
Positioning and Sizing Graphics Windows	17
Bringing a Graphics Window Forward on the Display	17
Putting a Title on a Graphics Window	18
Erasing a Graphics Window	18
Simple Graphical Displays	19
Chapter Overview	19
Simple Graphical Displays in IDL	19
Creating Line Plots	20
Customizing Graphics Plots	22
Modifying Line Styles and Thicknesses	22
Displaying Data with Symbols Instead of Lines	23
Displaying Data with Lines and Symbols	23
Creating Your Own Plotting Symbols	24
Drawing Line Plots in Color	24
Limiting the Range of Line Plots	25
Changing the Style of Line Plots	26
Plotting Multiple Data Sets on Line Plots	28
Plotting Data on Multiple Axes	29
Creating Surface Plots	29
Customizing Surface Plots	31
Rotating Surface Plots	31
Adding Color to a Surface Plot	32
Modifying the Appearance of a Surface Plot	34
Creating Shaded Surface Plots	34
Changing the Shading Parameters	35
Using Another Data Set For the Shading Values	35

Creating Contour Plots	36
Selecting Contour Levels	38
Modifying a Contour Plot	39
Changing the Appearance of a Contour Plot	39
Adding Color to a Contour Plot	41
Creating a Filled Contour Plot	42
Positioning Graphic Output in the Display Window	44
Setting the Graphic Margins	46
Setting the Graphic Position	46
Setting the Graphic Region	47
Creating Multiple Graphics Plots	47
Leaving Room for Titles with Multiple Graphics Plots	48
Non-Symmetrical Arrangements with !P.Multi	50
Adding Text to Graphical Displays	51
Finding the Names of Available Fonts	52
Adding Text with the XYOutS Command	52
Using XYOutS with Vector Fonts	53
Aligning Text	54
Erasing Text	54
Orienting Text	55
Positioning Text	55
Adding Lines and Symbols to Graphical Displays	56
Adding Color to Your Graphical Displays	57
Working with Image Data	59
Chapter Overview	59
Working with Images	59
Displaying Images	60
An Alternative Image Display Command	62
Scaling Image Data	62
Scaling Images into Different Portions of the Color Table	63
Displaying 8-Bit Images with Different Color Tables on 24-Bit Displays	64
Displaying 24-Bit Images	64
Displaying 24-Bit Images on 24-Bit Displays	65
Displaying 8-Bit Images on 24-Bit Displays	66
Automatic Updating of Graphic Displays When Color Tables are Loaded	66
Controlling Image Display Order	68
Changing Image Size	68
Changing Image Size in PostScript	69
Positioning an Image in the Display Window	69
Positioning Images with Normalized Coordinates	70
Reading Images from the Display Device	72
Obtaining Screen Dumps on 8-Bit Displays	72
Obtaining Screen Dumps on 24-Bit Displays	73
Reading a Portion of the Display	73

Table of Contents

An Alternative to TVRD	73
Basic Image Processing in IDL	74
Histogram Equalization	74
Smoothing Images	75
Removing Noise From Images	77
Enhancing the Edges of Images	77
Frequency Domain Filtering of Images	78
Building Image Filters	78
Graphical Display Techniques	81
Chapter Overview	81
Working with Colors in IDL	81
Using the Indexed versus the RGB Color Model	82
Static versus Dynamic Color Visuals	83
Specifying Colors on an 8-Bit Display	84
Specifying Decomposed Colors on a 24-Bit Display	84
Specifying Undecomposed Colors on a 24-Bit Display	86
Determining if Color Decomposition is On or Off	86
Obtaining Device Independent Colors	87
Loading Color Tables on a 24-Bit Display	88
Obtaining a Copy of the Color Table	89
Modifying and Creating Color Tables	89
Saving Your Own Color Tables	91
Creating Your Own Axis Annotations	92
Adjusting Axis Tick Intervals	92
Formatting Axis Annotations	92
Writing a Tick Format Function	94
Handling Missing Data in IDL	95
Setting Up a 3D Coordinate System in IDL	97
Setting Up a 3D Scatter Plot	98
Positioning the 3D Axes Through the Origin of a Plot	99
Combining Simple Graphical Displays	100
Animating Data in IDL	102
Setting Up the Animation Tool	103
Loading the Animation Buffer	103
Running the Animation Tool	103
Controlling the Animation	103
Saving Animation Pixmaps	104
Animating Other Types of Graphic Data	104
Gridding Data for Graphical Display	106
Delaunay Triangulation Method of Gridding	106
Spherical Gridding of Data	108
Using the Cursor with Graphical Displays	109
When Is the Cursor Position Returned?	110

Which Mouse Button Was Used with the Cursor? 111	
Annotating Graphics Output with the Cursor 111	
Drawing a Box 111	
Using the Cursor with Images 112	
Using the Cursor in Loops 113	
Erasing Annotation From the Display 114	
The “Exclusive OR” Method of Erasing Annotation 114	
The Device Copy Method of Erasing Annotation 116	
Drawing a Rubberband Box 118	
Graphics Window Scrolling 119	
Graphics Display Tricks in the Z-Graphics Buffer 120	
The Z-Graphics Buffer Implementation 121	
A Z-Graphics Buffer Example: Two Surfaces 121	
Make the Z-Graphics Buffer the Current Device 121	
Configure the Z-Graphics Buffer 122	
Load the Objects into the Z-Graphics Buffer 122	
Take a Picture of the Projection Plane 122	
Display the Result on the Display Device 122	
Some Z-Graphics Buffer Oddities 123	
Warping Images with the Z-Graphics Buffer 123	
Transparency Effects in the Z-Graphics Buffer 125	
Combining Z-Graphics Buffer Effects with Volume Rendering 126	
Reading and Writing Data in IDL 129	
Chapter Overview 129	
Opening a File for Reading or Writing 129	
Locating and Selecting Data Files 130	
Selecting File Names 130	
Selecting Directory Names 131	
Finding Files 131	
Constructing File Names 131	
Obtaining a Logical Unit Number 131	
Using Logical Unit Numbers Directly 132	
Allowing IDL to Manage Logical Unit Numbers 132	
Determining Which Files are Attached to Which LUNs 133	
Reading and Writing Formatted Data 133	
Writing a Free Format File 133	
Reading a Free Format File 134	
Rules For Reading Free Format Data 134	
Examples of Reading and Writing Free Format Files 136	
Reading a Simple Data File 136	
Writing a Column-Format Data File 137	
Reading a Column-Format Data File 137	
Creating a Template for Reading Column-Format Data 139	
Writing with an Explicit File Format 140	
A Few Common Format Specifiers 140	
Writing a Comma Separated Explicitly Formatted Data File 141	

Table of Contents

Reading a Comma Separated Explicitly Formatted Data File	141
Reading Formatted Data From a String	141
Reading and Writing Unformatted Data	142
Reading an Unformatted Image Data File	142
Writing an Unformatted Image Data File	143
Reading Unformatted Data Files with Headers	144
Problems with Unformatted Data Files	145
Accessing Unformatted Data Files with Associated Variables	145
Advantages of Associated Variables	146
Defining Associated Variables	146
Reading and Writing Files with Popular File Formats	147
Querying Image Files for Information	147
Creating a Graphic Display Program	149
Creating Color GIF Files	150
If the Display Depth is Eight	151
If the Display Depth is Greater than Eight	152
Writing the GIF File	153
Reading a GIF File	153
Creating Color JPEG Files	154
If the Display Depth is Eight	154
If the Display Depth is Greater than Eight	155
Writing the JPEG File	155
Reading a JPEG File	155
Creating Color TIFF Files	156
Reading Dicom Image Files	157
Using the IDLffDicom Object	157

Reading and Writing HDF Data161

Chapter Overview	161
Why Use the HDF Format?	162
Primary HDF Data Objects	162
HDF Application Programming Interface	163
Working with HDF Files	165
Opening HDF Files	165
Closing HDF Files	166
Determining the Number of Tags in an HDF File	166
Working with Scientific Data Set HDF Files	166
Optional Dimension Scales	167
Optional User-Defined Attributes	167
Optional Predefined Attributes	167
Opening HDF Files Containing Scientific Data Sets	169
Closing HDF Files Containing Scientific Data Sets	169
Creating or Selecting Scientific Data Sets	169
Creating a New SDS	169
Selecting an Existing SDS	170

Adding Attributes to Scientific Data Sets and HDF Files	170
Gathering Information about Scientific Data Sets	172
Gathering SDS Attribute Information	173
Adding Color Palettes to HDF Files	173
Examples of Reading and Writing HDF Files	174
Creating Hardcopy Graphics Output	175
Chapter Overview	175
Selecting the Graphics Hardcopy Output Device	175
Configuring the Graphics Hardcopy Output Device	176
Determining the Current Device Configuration	176
Common Device Command Keywords	177
Creating the PostScript File	178
Sending Graphics to the Hardcopy Device	179
Printing PostScript Files	180
Printing PostScript Files on Computers Running MacOS	181
Printing PostScript Files on a Windows Computer	181
Producing Encapsulated PostScript Output	181
Encapsulated PostScript Graphic Preview	182
Producing Color PostScript Output	182
Color and Gray Scale Images in PostScript	183
True-Color Images	183
Creating Quality Output on PostScript Devices	184
Similarities Between the Display and PostScript Devices	184
Differences Between the Display and PostScript Devices	185
Problem: PostScript Windows May Have a Different Aspect Ratio	185
Solution: Make the Aspect Ratios of Graphics Windows the Same Size	185
Problem: PostScript Devices Have a Higher Display Resolution	186
Solution: Don't Use Device Coordinates to Position Graphics	187
Problem: PostScript Devices Can Use Different Display Fonts	187
Solution: Take Care in Designing and Positioning Text	187
Problem: PostScript Devices Use Background and Plotting Colors Differently	189
Solution: Understand How PostScript Handles Background and Plotting Colors	190
Problem: PostScript Devices Often Have More Colors Than the Display Device	191
Solution: Be Sure to Scale Your Data Appropriately in PostScript	192
Problem: PostScript Devices Display Images Differently	193
Solution: Size Images with the TV Command	195
Calculating PostScript Offsets in Landscape Mode	198
Configuring the PostScript Device with PSConfig	198
Configuring and Using the Printer Device	200
Positioning Graphics with the Printer Device	202
Outputting Images with the Printer Device	203
Loading Colors in the Printer Device	204

Table of Contents

Color Loading Work-Arounds	205
IDL Programming Fundamentals	207
Chapter Overview	207
Writing an IDL Batch File	207
Writing a Main-Level IDL Program	208
Writing an IDL Procedure	209
Scope of Procedure and Function Variables	210
Creating a Positional Parameter	211
Defining Optional or Required Positional Parameters	212
Defining a Keyword Parameter	213
Using Keyword Abbreviations	213
Defining Optional Keyword Parameters	214
Is the Keyword Defined?	214
Handling Keywords with Binary Properties	215
Passing Undefined Keywords by Keyword Inheritance	216
Creating Output Parameters	217
Passing Information by Reference or by Value	217
Using Keyword Inheritance with Output Parameters	219
Is the Parameter Present?	220
Was the Parameter Used?	220
Writing an IDL Function	221
Square Bracket Notation and Function Calls	222
Reserving Function Names with the Forward_Function Command	223
Using Program Control Statements	223
True and False Expressions in IDL	223
Making Multiple Statements Appear As Single Statements	224
The IF...THEN...ELSE Control Statement	225
The Conditional Expression	226
The FOR Loop Control Statement	226
The WHILE Loop Control Statement	226
The REPEAT...UNTIL Loop Control Statement	227
The BREAK Control Statement	227
The CONTINUE Control Statement	227
The CASE Control Statement	227
The SWITCH Control Statement	228
The GOTO Control Statement	229
Error Handling Control Statements	229
The On_IOError Control Statement	229
The On_Error Control Statement	230
The Catch Control Statement	230
Error Handling Hierarchy	231
Reporting Errors	232
Generating Errors	233
Tracing Errors	234
Compiling and Running IDL Program Modules	235
Rules for Compiling IDL Program Modules Automatically	236

Structuring Program Files	236
Special Compilation Commands	237
Writing an IDL Graphics Display Program	239
Chapter Overview	239
The HistoImage Program	240
Writing the Procedure Definition Statement	240
Writing the Error Handling Code	242
Checking for Positional and Keyword Parameters	242
Checking for the Image Positional Parameter	243
Checking for Keyword Parameters	243
Loading the Program Colors	246
Preparing to Draw the Graphics	246
Calculating Graphic Positions in the Window	246
Changing Character Size According To Window Size	247
Calculating the Image Histogram	248
Drawing the Graphics	248
Drawing the Histogram Plot	248
Drawing the Color Bar	251
Drawing the Image Plot	251
Working Around a Printer Device Bug	252
Compiling and Testing the Program	252
Reviewing the HistoImage Program's Advantages	253
The HistoImage Program is Device Independent	255
Using HistoImage in a "Smart" Resizeable Graphics Window	256
Writing a Widget Program	259
Chapter Overview	259
The Structure of Widget Programs	259
How Do Widget Programs Respond to Events?	261
Writing the Widget Definition Module	261
The Advantage of Mistakes	261
Typing the Code	262
Defining and Creating the Program's Widgets	263
Creating the Top-Level Base Widget	264
Creating Buttons for the Menu Bar	265
Creating the Graphics Window for the Program	265
Realizing the Widgets on the Display	266
Making the Draw Widget the Current Graphics Window	266
Displaying the Graphics in the Draw Widget Window	266
Storing Information Required to Run the Program	267
Using Pointer Variables	268
Using Pointers in the Info Structure	271
Using Widget User Values to Store Program Information	271
Practicing Good Memory Management	272

Table of Contents

Creating the Event Loop and Registering the Program	275
Running the Program	276
Writing the Event Handler Modules	276
Common Fields in Event Structures	277
Event Handler Functions	278
Associating Event Handlers with Widgets	279
Writing the Quit Button Event Handler	279
Writing the Resizeable Graphics Window Event Handler	280
Writing the Cleanup Procedure	282
Running the Program	283
Recovering From Program Errors	283
Adding Color Protection	284
Saving the Color Vectors	285
Setting Up Keyboard Focus Events	285
Modifying the Histo_GUI_TLB_Events Event Handler	286
Buffering the Graphic Display for Smoother Output	287
Widget Programming Techniques	293
Chapter Overview	293
Adding Image Processing Capability	294
Building the Pull-Down Menu Widgets	294
Writing the Event Handler for the Pull-Down Menu	295
Limitations of the Event Handler as Written	296
Implementing an Undo Capability	298
Adding Color Controls to the Program	301
Building the Pull-Down Menu Widgets	301
Writing the Drawing Colors Event Handler	301
Writing the Image Colors Event Handler	305
Communicating in XColors via Widget Events	306
Importance of Group Leaders	310
Adding File Output Functionality	312
Building the Pull-Down Menu Widgets	312
Writing the File Output Event Handler	313
Creating the GIF File	315
Creating the JPEG File	315
Creating the TIFF File	316
An Alternative Way of Creating GIF, JPEG, and TIFF Files	316
Creating the PostScript File	317
Adding Printer Functionality	320
Creating the Print Pull-Down Menu	320
Writing the Print Event Handler	320

Creating Dialog Form Widgets 325

Chapter Overview **325**

Creating a Modal Dialog Form Widget **325**

 A Blocking Widget Program **326**

 A Modal Widget Program **327**

 Writing a Modal Dialog Form Widget Definition Module **327**

 Defining a Modal Top-Level Base **328**

 Defining Other Widgets **329**

 Storing Collected Information in Modal Dialogs **330**

 Creating the Info Structure **331**

 Creating a Blocked Widget **331**

 Returning from the Block **331**

 Writing the Modal Dialog Event Handler Modules **332**

 Error Handling **333**

 Testing the Modal Dialog Form Widget Program **335**

 Using FSC_InputField for Program Input **335**

 Adding an Open Image Capability to the Histo_GUI Program **337**

 Adding an Open Button **337**

 Writing the Open Image Event Handler **338**

Creating a Non-Modal Widget Dialog **340**

 Writing a Non-Modal Dialog Widget Definition Module **341**

 Notifying Widgets of Program Events **342**

 Writing the Non-Modal Dialog Event Handler Modules **343**

 Sending Events to Other Widgets **344**

 Testing the ReadImage Program **345**

 Writing the Read Image Event Handler **345**

Creating Graphics Display Objects 349

Chapter Overview **349**

A Quick Object Overview **349**

 The Idea of Data Encapsulation **350**

 Creating Objects **350**

 Invoking Object Methods **351**

 Destroying Objects **353**

Creating a New Object Class **353**

 Defining the Object Class **353**

 Structure Review **353**

 Automatic Structure Definition **355**

 The BoxImage Class Definition **355**

 Creating Object Lifecycle Methods **356**

 Creating the Init Method **357**

 Creating the Cleanup Method **360**

 Creating the Specific Instance of the Object **360**

 Lifecycle Methods Must Be Defined When the Object is Created **361**

 Common Problems When Creating Objects **361**

 Initializing the Object Using Parameters **361**

 Creating the Display Method **362**

Table of Contents

Creating Methods to Set and Get Object Properties	364
Set Property Methods	365
Get Property Methods	367
Creating Methods to Work with Colors in Objects	370
Creating Methods to Extend Object Functionality	372
Object Inheritance	374
Defining the Subclass Object	375
Creating Subclass Lifecycle Methods	376
Attaching Methods to Superclass Objects	378
Overriding Superclass Methods	378
Creating New Methods for the SubClass Object	380
Object Polymorphism	380
Testing the HistoImage Object	386
Appendix A: Widget Event Structures	389
Event Structure Definition	389
Common Field Definitions	389
Basic Widget Event Structures	389
Base Widget Event Structure	389
Button Widget Event Structure	390
Draw Widget Event Structure	390
Dropdown Widget Event Structure	390
Label Widget Event Structure	390
List Widget Event Structure	390
Slider Widget Event Structure	391
Table Widget Event Structure	391
Character Insertion Event	391
String Insertion Event	391
Delete String Event	391
Text Selection Event	391
Cell Selection Event	391
Row Height Changed Event	392
Column Width Changed Event	392
Invalid Data Event	392
Text Widget Event Structure	392
Character Insertion Event	392
String Insertion Event	392
Delete String Event	392
Text Selection Event	392
Compound Widget Event Structures	393
CW_Animate Event Structure	393
CW_Arcbal Event Structure	393
CW_BGroup Event Structure	393
CW_Clr_Index Event Structure	393
CW_Color_Set Event Structure	393
CW_DefROI Event Structure	393
CW_Field Event Structure	393
CW_Form Event Structure	393

CW_FSlider Event Structure	393
CW_Orient Event Structure	394
CW_PDMenu Event Structure	394
CW_RGBSlider Event Structure	394
CW_Zoom Event Structure	394
FSC_InputField Event Structure	394
Widget Program Event Structures	394
XColors Event Structure	394
ReadImage Event Structure	395
Other Widget Event Structures	395
Keyboard Focus Events	395
Kill Widget Request Events	395
Widget Timer Events	395
Widget Tracking Events	395
Appendix B: Data File Descriptions	397
Appendix C: IDL Program Code	399
IDL Example Programs	399
BoxImage__Define Object Program	399
HDFRead Program	406
HDFWrite Program	407
HistoImage Program	409
Histo_GUI Program	412
HistoImage__Define Object Program	423
OpenImage Program	428
ReadImage Program	431
Index	435

Acknowledgements

♦ Discovering the Possibilities ♦♦♦



Preface

Preface to the First Edition

Writing a book must surely be one of the most difficult, solitary, and lonely things I have ever done. And yet, paradoxically, it cannot be done at all without the help and support of a great many people. Foremost among these is my wife, Carol. It is not possible to write a book, make a living, and pay adequate attention to hearth and home, all at the same time. Carol has made it possible—in more ways than you can imagine—to have this obsession realized. I am deeply grateful to her and to our three sons for the time I borrowed from their lives. Merry Christmas! I am finally home.

Many people have read earlier versions of this book and have offered their comments and suggestions. I wish to thank all of them for their help. I am particularly indebted to Dick Jackson of the National Research Council of Canada who volunteered a great many hours of his own time to read the entire manuscript from start to finish and offer extensive valuable suggestions. This is a much better book as a result of his efforts. I am also thankful to many fine folks at Research Systems and on the IDL newsgroup who have patiently answered many of my questions about IDL and have been supportive of my work. The errors that remain in the book are entirely my own.

I owe a special debt of gratitude to David Stern, the founder of Research Systems and the original creator of IDL. I spent five of the best years of my professional life working for David. I am deeply grateful for the freedom David gave me to pursue my overriding interest, which was to teach people how to use IDL. Working with IDL is the best job I have ever had.

I don't think this book would have ever happened without a thin volume of stories by Barry Lopez, called *River Notes/Desert Notes*. I don't know why this book speaks to me in the way it does, but I know I owe much of my professional life and development to its messages. Stories tell us who we are and connect us to the world we live in. I often tell his wonderful story, *Directions*, in my IDL courses. It is a powerful symbol to me of what I am about both personally and professionally. Thank you, Mr. Lopez, for guiding and nourishing me with the mystery of your stories.

Finally, I wish to thank the hundreds of people who have attended my IDL programming courses over the past six years. I learned almost everything I know about IDL from you. Thanks for making those workshops warm, safe places where we could all make mistakes and learn from one another. This book is written especially for you.

Fort Collins, Colorado
Christmas Eve, 1997

Preface to the Second Edition

The problem with writing a book is that it tends to crystallize ideas that are current at the time the book is written and set them—if not in stone—then certainly in type, which is almost as hard a substance. But programming ideas are such fluid creatures that an author finds himself looking at what he created with a skeptical eye about five minutes after the book is sent off to the printer, and it just gets worse from there. After almost three years you begin to think to yourself, “Was I in my right mind when I wrote that nonsense!?”

Apparently so, because people keep buying the book and telling you how wonderful it all is. But, finally, embarrassment overcomes flattery and you set out on that long, lonely journey again.

The biggest change that has happened in the past three years is that most of us have moved from 8-bit displays with 256 colors to 24-bit displays with millions of colors. And while I have been able to tweak the book from one printing run to the next to “kind of” keep up with the changes, I realized not too long ago that I was really writing my programs in a different way these days. So while the chapters on basic IDL commands have not changed radically, the last five programming chapters have been completely re-written to reflect my approach to writing programs that more often than not live in a 24-bit world, but have to work in an 8-bit world, too.

What this book still lacks, at least to my mind, is a comprehensive review of objects and object-oriented programming. I may add it some day, or I may write a separate book on the subject. I seem to vacillate back and forth from one day to the next. I know in my own programming I almost never write a program that doesn’t include an object of one kind or another. But there is great reluctance on the part of IDL users who are not full-time programmers (the vast majority) to write object programs. And I understand this reluctance, because it seems to take you out of the realm of “scientist” and into the realm of “programmer”. Many of us are naturally suspicious of identifying with that crowd.

For those of you in this reluctant majority, the good news is you can still find 80 percent of what you are ever going to want to do with IDL in the pages of this book. The bad news is you are missing out on some techniques that could make a huge difference in the quality of the programs you write. We will pull you along, eventually. There is just too much of value there to ignore forever.

Thanks again to my ever persevering family, and especially to my wife, Carol, who provides support in so many ways. And I couldn’t make a living, let alone write a book, without the people who attend my IDL programming courses each year. This book wouldn’t be possible without you. But my special thanks go to the folks who frequent the IDL newsgroup (`comp.lang.idl-pwave`). I have never met a friendlier, more helpful community of users in my life. I greatly appreciate your humor, your insight, and your unstinting devotion to sharing your IDL knowledge with beginners and experts alike. This work wouldn’t be nearly as much fun without you.

Now, if you will excuse me, I’m going to make myself a gin and tonic and sit out on the back porch of our new house. I’ve been promising my wife all summer I would.

Fort Collins, Colorado
October, 2000

Chapter 1

◆ Discovering the Possibilities ◆◆◆



Getting Started

Chapter Overview

The purpose of this chapter is to explain why I wrote this book, what you can expect to get out of reading it, and to give you information that will make it easier to work through the IDL programming examples you find here. Specifically, you will learn:

- How this book is organized
- How to use this book
- How to download and organize the files that come with this book
- How to use variables, keywords, and commands in IDL
- How to create and work with vectors and arrays in IDL
- How to work with graphics windows in IDL

Philosophy Behind this Book

This book grew out of many years of teaching scientists and engineers to use and program IDL (Interactive Data Language), most of the time while working for Research Systems, the developers of IDL. As I answered question after question, I realized most questions fall into a handful of broad categories. The truth is, most of us want to do pretty much the same things with IDL. We want to analyze and display our data, write efficient programs to solve our scientific problems, and—most of all—get our work done quickly. What most of us do *not* want to do is read computer software manuals.

IDL is a big software program that is getting bigger every day. It comes with an impressive (and heavy) amount of documentation, which no one I know wants to read. IDL can be intimidating even to experienced users, let alone to someone just starting out to learn its mysteries. This book is meant to bring IDL under control. It is meant to teach you 80 percent of what you absolutely need to know to work with IDL on a daily basis. And, most importantly, it is meant to do that with examples that are easy to understand. More than anything, it is a book that *shows* you how to work with IDL.

I envision the audience for this book as people just starting to use IDL and, more particularly, people who have had to learn to use IDL on their own. Learning IDL well is a long process. Most of us don't get the time in our jobs to do it properly. I wanted to write a book that can put IDL in context for both groups of people. To that end, this

book is an overview of the essential elements of IDL for people who don't like to read manuals and who learn best by example. It is a compendium of IDL programming tips and techniques that can only be learned by experience. Essentially, this is the book I wish I had when I was learning to use IDL!

Using this Book

I've tried to make each chapter in the book self-contained so that you can pick the book up and turn to any chapter to learn what is most pressing for you. But I have also arranged the chapters in the book in more or less the order I teach the material in IDL programming courses. If you are just starting to learn IDL, it probably makes sense to start at the beginning and work through the material in the order it is presented. The material in the later programming chapters builds on concepts and techniques that were developed in earlier chapters.

Required Version of IDL

I assume you will be using the latest version of IDL in this book. This was version 5.3.1 at the time this book was written. People with earlier versions of IDL will probably be able to complete most of the programming examples in the book, but I have made no effort to make the example programs that come with the book (see below) compatible with earlier versions of the software. The IDL news group (*comp.lang.idl-pvwave*) is a good source of help if you are having difficulties modifying your programs to run in earlier versions of IDL.

If you need to upgrade your software, you can find information about Research Systems and their local IDL distributors, including how to upgrade your software, on the Research Systems home page at this World Wide Web location:

<http://www.rsinc.com/>

Working with Colors in the IDL Session

The programming examples in the book have been written to work on either 8-bit or 24-bit color displays. If you ever have trouble getting your colors to be what you expect them to be, try setting the *Decomposed* keyword of the *Device* command to 0. This fixes about 80 percent of the color problems you are likely to encounter.

IDL> Device, Decomposed=0

Realize that if you change the current color table on a machine with a 24-bit display and you are working at the IDL command line, you will almost certainly need to re-type the graphic display command to see the new colors take effect. This is normal and is a direct effect of how 24-bit color works. You will see how to deal with this more effectively later in the book.

You can determine the depth of your color display by typing these two commands:

IDL> Device, Get_Visual_Depth=thisDepth & Print, thisDepth

Style Conventions Used in the Book

I try to use a consistent style throughout the book so that I don't confuse you about the function or purpose of the text. First, IDL commands that you should type at the IDL command line or into an IDL editor window are always set in *Courier* type face:

Surface, data

Commands that you should type at the IDL command line are preceded by the IDL command prompt, **IDL>**, like this:

```
IDL> Surface, data
```

Other IDL commands will be typed in editor windows. You can use the editor of your choice or the editor that is supplied with IDL. It is up to you.

Capitalization

I use a particular style of capitalization for IDL commands in this book. This style is completely arbitrary. IDL is case insensitive, except for commands that interact with the operating system (e.g., the file names of commands that open files will be case sensitive on UNIX machines) and when it is performing string comparisons. The capitalization is meant to help you remember command and keyword names and to give you a visual clue as to the function of words on the command line.

I capitalize the first letter of all IDL commands and keywords. In addition, any letter that may serve as a mnemonic is also capitalized. For example:

```
Surface, data, CharSize=2.0, Color=180
XLoadCT
Widget_Control, tlb, Set_UValue=info, /No_Copy
```

I do not capitalize the first letter of variable names, although I may capitalize subsequent letters in variable names that may be compound words. For example:

```
data = FIndGen(11)
buttonValue = thisValue
ptrToData = Ptr_New()
```

I completely capitalize IDL reserved words. For example:

```
REPEAT test UNTIL
FOR j=0,10 DO BEGIN
ENDWHILE
```

You may use any capitalization you like when you type the commands at the IDL command line or into a text editor.

Comments

Anything to the right of a semi-colon on an IDL command is treated as a comment by IDL and is ignored by the IDL interpreter. In general, I try to write comments on their own line in an IDL program, usually set off by a blank space before and after it and indented three spaces from the line it is commenting on. For example:

```
; This is the loop part of the program.

FOR j=0,10 DO BEGIN
    data = j*2
    count = count + j
ENDFOR
```

Occasionally you will see a comment on the end of a command line. I do this especially when I am documenting the fields of an IDL structure variable. For example:

```
info = {r:r, $      ; The red color vector
        g:g, $      ; The green color vector
        b:b }       ; The blue color vector
```

Line Continuation Characters

The line continuation character in IDL is the dollar sign, \$. This indicates that the IDL command is continued on the next command line. (See the example above.) You will see a lot of line continuation characters in the IDL commands in this book. My advice, especially for those commands that you should type at the IDL command line, is to ignore the line continuation characters (leave them out) and just keep typing the IDL command on the same command line. For example, you might type the command above like this:

```
IDL> info = { r:r, g:g, b:b }
```

This will make it much easier for you to re-type the command if you make a typing mistake or if you need to modify the command later.

There are occasions in this book when you should type the IDL commands *exactly* as they appear in the book. I'll let you know when this is the case. This will almost always be when I want you to type a FOR loop at the IDL command line. It is extremely tricky to write multiple line commands at the IDL command line. You have to make the IDL interpreter think the commands are a single command. This requires specific use of line continuation (\$) and multiple command (&) characters on the IDL command line.

IDL Programs and Data Files Used in the Book

A number of IDL program files have been prepared for you to use with this book. The IDL program files always have a *.pro* file extension.

Installing the Program Files

This book assumes that you will create a subdirectory named *coyote* and will put all the program files there. Often the *coyote* subdirectory is a subdirectory of the main IDL directory (i.e., the directory pointed to by the *!Dir* system variable inside of IDL), but it does not have to be in this directory. You can create it anywhere. Your IDL *home directory* (see below) is another good place to put it. It is a good idea not to work directly in the *coyote* subdirectory, but to copy program files from here into your current working directory as you need them. This way, you always have the original files available to you should you need to refer to them.

If you choose not to create a *coyote* subdirectory, then the programs used with the book assume that the program files are located in your current directory. This directory is normally the one in which you invoked IDL or, in the case of IDL on a PC or Macintosh computer, the home directory specified in the *Startup* dialog of the *Preferences* menu.

Determining Your IDL Home and Current Directory

If you are not sure what your IDL home directory is, start IDL and type these commands as the first commands in your IDL session:

```
IDL> CD, Current=homeDirectory  
IDL> Print, homeDirectory
```

Your current directory, which does not have to be your home directory, can be found at any time during your IDL session with the same commands:

```
IDL> CD, Current=currentDirectory  
IDL> Print, currentDirectory
```

 It is probably a good idea *not* to use the main IDL directory (e.g., the *IDL53* directory in the case of Windows IDL) as your working directory. It is too easy to delete a file that may be important to you!

Downloading the Program Files Used with the Book

The program files are available via the Internet or via anonymous ftp. If you are using an Internet browser, follow the links starting at the *Coyote's Guide to IDL Programming* web page. Its World Wide Web address is:

```
http://www.dfanning.com/
```

If you are using anonymous ftp, the files can be found via an Internet browser at:

```
ftp://ftp.dfanning.com/pub/dfanning/outgoing/coyote2nd
```

Copy all program files in text or *ASCII* mode. If you like—and your computer can uncompress a zip file archive—you can copy all the program and text files at once by copying the *coyotefiles.zip* file. This is a zip file archive of the program files you need.

Make Sure Your Coyote Directory is on the IDL Path

No matter where you create the *coyote* subdirectory or store the book files, you want to make sure that directory is on your IDL path. The path is given by the *!Path* system variable in IDL. You will learn more about this system variable later, but for now it is enough to know that this is a list of subdirectories that IDL searches to resolve commands it doesn't recognize. You can see your current IDL path by printing this system variable:

```
IDL> Print, !Path
```

Notice that if you are on a PC these subdirectories are separated by semi-colons; on a Macintosh or VMS machine, they are separated by commas; and on a UNIX machine, they are separated by colons.

You want to add the *coyote* directory to the IDL path by typing the command *AddPath* from within the *coyote* directory. (If you didn't create a *coyote* directory, you can type the *AddPath* command from within the directory that contains the downloaded book files.) Use the *CD* command in IDL to change to the appropriate directory. For example, if your *coyote* directory is a subdirectory of your IDL home directory and the home directory is where you are currently located, you can add the *coyote* directory to the IDL path by typing these commands:

```
IDL> CD, 'coyote'  
IDL> AddPath
```

You will want to move into the *coyote* directory (or the directory where your book files are located) and run the *AddPath* program each time you run IDL and work with this book. You might think of adding the command to your IDL start-up file. Or, you will want to add the *coyote* directory to your path permanently. (This is done differently depending upon your operating system and IDL file configuration. See the IDL on-line help for information on setting your *!Path* system variable. If you are using the IDL Development Environment (IDLDE), you can set the *!Path* system variable from the *File->Preferences->Path* tab.)

Copying the Data Files

The data files used with the book are already in the IDL distribution. If you prefer to collect them in a single directory, you can copy them into any directory you like. The *coyote* directory is a likely candidate. To do this, use the program *CopyData*, which is one of the files you just downloaded. Go into the *coyote* directory (or the directory where your book files are located) and just type *CopyData*, like this:

```
IDL> CD, './coyote'  
IDL> CopyData
```

The data files will be collected from their various locations and copied into your current directory. There is a list of the data files that will be used in the book, along with their data types and sizes, in “Appendix B: Data File Descriptions” on page 397.

Obtaining Additional Help

If you have difficulty installing the program files or if you need help with some other aspect of IDL programming, check the *Coyote’s Guide to IDL Programming* web page. You will find information there about this book and about IDL programming in general. If worst comes to worst, you will also find a form there that will allow you to contact me directly. The World Wide Web address of Fanning Software Consulting and the *Coyote’s Guide to IDL Programming* is:

<http://www.dfanning.com/>

Working with IDL Commands

This book is meant to be a *doing* book. I prefer that you read it sitting in front of a computer rather than in front of a fire. I want you typing commands and seeing what happens. For that reason, most of the commands in the first half of this book are meant to be typed at the IDL command line. (If you wish to keep a record of your commands as you type them, you can create a journal file to record them. See “Creating Command Journals” on page 8 for additional information.)

IDL has evolved tremendously as a programming language in the 14 years I’ve been working with it. But there is still a lot to be gained from learning how to use IDL from the command line. In particular, you learn to figure things out, to try things, to experiment with your data. I call it “learning by noodling around.” I think it is one of the best ways to use IDL.

So here is what you need to know to get started. First, you will see a lot of commands like this in the book:

```
Contour, peak, lon, lat, XStyle=1, YStyle=1, /Follow, $  
    Levels=vals, C_Labels=[1,0,1,0,0,1,1,0]
```

It helps if you know what you are looking at.

Anatomy of an IDL Command

The word *Contour* in the command above is the name of the IDL command or program you wish to run. It must be spelled out in its entirety. Some command names can be quite long, but no shortcuts are allowed. The words *peak*, *lon*, and *lat* in this command are *variables*. They are used to pass information into or out of the command or program. The words *XStyle*, *YStyle*, *Follow*, *Levels*, and *C_Labels* are *keywords*. Keywords are by convention optional parameters to the command. Like variables, they are used to pass information into and out of the command or IDL program.

Positional Parameters

The three variables *peak*, *lon*, and *lat* in the command above are also called *positional parameters*. In this particular instance, these positional parameters are *input* variables (i.e., they are bringing data into the command), but you cannot tell this by looking at them. They could just as easily be *output* variables. (Or they could be *both* input and output variables, for that matter.) The command line syntax is exactly the same. You will only be able to tell by context and by reading the published documentation for the command or program.

A positional parameter has a defined sequence or order to the right of the command name. (Note that *keyword parameters*, discussed below, do not affect positional parameter order.) In this case, the variable *peak* must be to the right of the command *Contour* and to the left of the variable *lon*, for example. The variable *lon* must be to the right of *peak* and to the left of *lat*, and so on. You cannot leave out, say, the second positional parameter and specify the first and third.

For example, these two commands are incorrectly formatted and will cause errors. The first because the order of the positional parameters is changed, and the second because the second positional parameter is not present.

```
Contour, lon, peak, lat, XStyle=1, YStyle=1, /Follow, $  
    Levels=vals, C_Labels=[1,0,1,0,0,1,1,0]  
  
Contour, peak, , lat, XStyle=1, YStyle=1, /Follow, $  
    Levels=vals, C_Labels=[1,0,1,0,0,1,1,0]
```

Positional parameters are often required parameters to the command, but they do not have to be. For example, in the correct command above *peak* is a required parameter to the *Contour* command, but *lon* and *lat* are optional positional parameters. Again, you will know this by reading the published documentation for the command.

Keyword Parameters

XStyle, *YStyle*, *Follow*, *Levels*, and *C_Labels* are *keyword parameters*. Unlike positional parameters, keyword parameters can come in any order to the right of the command name. They can even come in the midst of positional parameters without affecting the relative positions of those parameters. In other words, keyword parameters are not counted like positional parameters. This is a valid construction of the *Contour* command above:

```
Contour, peak, Levels=vals, lon, XStyle=1, YStyle=1, $  
    /Follow, Levels=vals, lat, C_Labels=[1,0,1,0,0,1,1,0]
```

By convention keyword parameters are optional parameters. Like positional parameters they can be input parameters or output parameters to the command. You will know by context and by reading the documentation for the command.

Notice the way in which the keywords are used in the command above. Keywords can be set to a particular value (e.g., *XStyle=1*), to a variable (e.g., *Levels=vals*), to a vector of values (e.g., *C_Labels=[1,0,1,0,0,1,1,0]*), and even set with a slash character (e.g., */Follow*).

Consider the latter syntax more closely. Some keywords have a binary quality. That is, they are either on/off, yes/no, true/false, 1/0, etc. You often find these keywords being “set” or “turned on” by the syntax */Keyword*. The syntax */Keyword* is identical to (means the same as) the syntax *Keyword=1*. No more and no less.

In fact, the *Contour* command above could have been written like this:

```
Contour, peak, Levels=vals, lon, /XStyle, /YStyle, $  
    /Follow, Levels=vals, lat, C_Labels=[1,0,1,0,0,1,1,0]
```

This command means the same thing as the command above. The reason the command was not written like this, is that it might falsely imply that the *XStyle* and *YStyle* keywords have a binary quality (i.e., they are either on or off), which they don’t. They can be set to other values besides 0 and 1.

IDL Procedures and Functions

This particular command, the *Contour* command, is an IDL procedure. IDL commands will be either procedures, like this one, or functions. Here is an example of an IDL command, the *BytScl* command, that is a function:

```
scaled = BytScl(image, Top=199, Min=0, Max=maxValue)
```

Notice the difference between the *Contour* procedure command and the *BytScl* function command. First of all, in the function command the positional parameters and keywords are enclosed in parentheses. In the procedure command the parameters and keywords are simply listed on the command line. But the most important difference is that the function command explicitly returns a value, which is placed in a variable on the left-hand side of the equal sign. This is the fundamental difference between a function command and a procedure command in IDL.

Function commands *always* explicitly return a value that must be assigned to a variable. The return value of a function may be any kind of IDL variable, including scalars, vectors, and structures. In this case, the return value, *scaled*, is a byte array of the same dimensions as the *image* positional parameter.

Sometimes you will see a function command and a procedure command written together. For example, consider these two commands:

```
scaled = BytScl(image, Top=199, Min=0, Max=maxValue)
TV, scaled
```

The first command is a function command and the second command is a procedure command that uses as its positional parameter the return value of the function. It would not be unusual in IDL to see these two commands written like this:

```
TV, BytScl(image, Top=199, Min=0, Max=maxValue)
```

In this case, the *BytScl* command must be evaluated first and a value returned. That return value is used as the positional parameter of the *TV* command.

It will probably take a while to become familiar enough with the various IDL commands so that you know immediately which is a procedure and which is a function, but try to remember this: if you are looking for *one* value from a command, the command is probably a function. You will learn how to write IDL procedures and functions later in this book.

Help with IDL Commands

IDL comes with an extensive on-line help system that can give you extremely helpful information about IDL commands and their parameters. On-line help is accessed by simply typing a question mark at the IDL command prompt or by selecting the *Help* menu item from the IDL Development Environment pull-down menu. Almost all the information in the IDL documentation set is available on-line. Check the *docs* subdirectory of the main IDL directory if you are having trouble locating what you need. To access the IDL on-line help system, simply type a question mark at the IDL prompt, like this:

```
IDL> ?
```

Creating Command Journals

You may wish to save a journal or record of the commands you type at the IDL command line. If so, you can create a *journal file*. A journal file is an IDL batch file (see “Writing an IDL Batch File” on page 207 for more details). A journal file is opened in IDL by using the *Journal* command and specifying the name of the file you wish to open. The file will be a new file, open for writing. There is no way to append to a journal file from the IDL command line. To open a journal file named, for example, *book_commands.pro*, type this:

```
IDL> Journal, 'book_commands'
```

All subsequent commands that you enter at the IDL command line will go into this journal file.

```
IDL> a = [3, 5, 7, 3, 6, 9]
IDL> Help, a
IDL> Plot, a
```

When you want to close the journal file, just type the *Journal* command again, all by itself at the IDL command line, like this:

```
IDL> Journal
```

The journal file is a simple ASCII text file that you can edit, if you like, with any text editor, including the editor built into IDL Development Environment. When you want to replay the commands in the journal file, use the @ sign as the first character on the IDL command line. For example, to replay the commands in the *book_commands.pro* file above, type this:

```
IDL> @book_commands
```



Be sure to give each journal file you create a unique name. You cannot append to journal files, so if you open a second file with the same name as the first many operating systems will simply overwrite the first journal file without warning.

If you would like to have a unique journal file name each time you wanted a journal file, you could write an IDL program like this:

```
PRO Journal_Unique
Journal, String('journal_', Bin_Date(SysTime()), '.pro', $
    Format='(A, I4, 5I2.2, A)')
END
```

Then, instead of typing *Journal*, you can type *Journal_Unique* to open a journal file with a unique name. This file has already been written for you and is one of the files you downloaded to use with this book.

Creating Variables

You will be creating many variables in this book. It will help if you know a little about them before you get started. Variable names must start with a letter. They can include other letters, digits, underscore characters, and dollar signs. A variable name may have up to 255 characters. A convention used in this book is to make the initial letter in variable names lowercase. Here are some valid variable names:

```
ptrToData
image2
this_image
a$handle
```

Variables have two important attributes: a *data type*, and an *organizational structure*. Data type refers to the kind of data it is. There are 14 basic data types in IDL (as of IDL 5.3). In Table 1 you see each basic type, the size of each variable type in bytes, the way a variable of that type can be created, and the name of the IDL function that can convert a variable to that data type. In addition to a data type, a variable has an organizational structure. Valid organizational structures are *scalars* (i.e., single values), *vectors* (really a one-dimensional array), *arrays* (of up to eight dimensions), and *IDL structures* (a type of organization that can contain variables of various data types and organizational structures in separate compartments called *fields*).

As you will see, IDL is a program that excels at working with vector or array data, so there are a number of built-in IDL commands for creating vectors and arrays of the different data types. In particular, there are functions for creating arrays of the proper data type in which each element is initialized to zero, and there are functions for creating arrays of the proper data type in which each element is initialized to its own

index in the array. You see a list of these functions in Table 2. For example, to create a 100 by 100 byte array of zeros, you can type this:

```
IDL> array = ByteArr(100,100)
```

To create a vector of 100 floating point values ranging in value from 0 to 99, you can type this:

```
IDL> vector = FIndGen(100)
```

You will see many ways to use these IDL functions in this book.

Data Type	Size (bytes)	Variable Creation	Data Type Function
Byte	1	var = 0B	thisVar = Byte(variable)
16-Bit Signed Integer	2	var = 0	thisVar = Fix(variable)
32-Bit (Long) Signed Integer	4	var = 0L	thisVar = Long(variable)
64-Bit Signed Integer	8	var = 0LL	thisVar = Long64(variable)
16-Bit Unsigned Integer	2	var = 0U	thisVar = UInt(variable)
32-Bit (Long) Unsigned Integer	4	var = 0UL	thisVar = ULong(variable)
64-Bit Unsigned Integer	8	var = 0ULL	thisVar = ULong64(variable)
Floating Point	4	var = 0.0	thisVar = Float(variable)
Double-Precision Floating	8	var = 0.0D	thisVar = Double(variable)
Complex	8	var = Complex(0.0, 0.0)	thisVar = Complex(variable)
Double-Precision Complex	16	var = DComplex(0.0D, 0.0D)	thisVar = DComplex(variable)
String	0-32767	var = '' or var = " "	thisVar = String(variable)
Pointer	4	var = Ptr_New()	None
Object	4	var = Obj_New()	None

Table 1: *The 14 basic data types in IDL. Also shown is the size of the variable, the way a variable of that type can be created, and the IDL function that can be used to cast or coerce a variable to that data type.*

Variable Attributes Change Dynamically

One of the most powerful properties of IDL is that most of its commands can work on data of any data type or organizational structure. This is so because IDL has the ability to change the data type and organizational structure of a variable at run time. (Like many powerful things in the world, this ability to change variable attributes dynamically also has the potential to be extremely dangerous! You need to exercise care to be sure you know what kind of data you are working with.) For example, it is essentially pointless to initialize variables in IDL (like you might in a Fortran or C program), because the data type of that variable can so easily change. Consider this example:

```
num = 3           ; Initialize NUM as a scalar integer.
num = num * 5.2   ; Variable NUM changes to a float!
```

Here the variable *num* is initialized as an integer and it gets dynamically changed to a floating point value as the result of the mathematical operation and redefinition of its

value. This is because IDL “promotes” variables to the data type that preserves the most accuracy in mathematical calculations. When *num* is redefined (on the left hand side of the equal sign) it is promoted to a float to maintain the accuracy of the floating point calculation on the right hand side of the equal sign.

Consider this example:

```
result = 4 * x
```

In this case, it is impossible to know what data type and organizational structure the variable *result* will have because you know nothing about the variable *x*. In fact, *result* will depend almost completely on the data type and organizational structure of variable *x*. If *x* is a floating-point vector of 10 elements, *result* will also be a floating-point vector of 10 elements. If it is a long-integer array of size 100 by 200, *result* will be the same. Note that if *x* has a date type of byte, that *result* will always have a data type of integer. (Organizational structure doesn’t matter, in this case.) This is a result of the multiplication by an integer value.



Remember that the expression on the right-hand side of the equal sign is always evaluated before a data type and organizational structure can be assigned to the variable on the left-hand side of the equal sign. IDL will promote variables to the data type that maintains the most precision in evaluating the expression

Be Careful With Integer Variables

I want to say just a quick word about integer variables so you stay out of trouble using them. There are two common ways to get into trouble. The first involves integer math. Consider this example:

```
result = 12/5
```

You may expect *result* to have a value of 2.4 and be a floating point value, but it is not. It has a value of 2 and is an integer. Can you think of the reason why? Right, the two numbers on the right-hand side of the equation are both integers. This is an example of integer division. In this case, it probably won’t be too hard to find your error, but sometimes this problem can be more subtle.



Suppose, for example, you wanted to know the aspect ratio of an IDL graphics window. You know that the size of the window (in pixels or integer values) is stored in two system variables. You might write your IDL code like this:

```
aspect = !D.X_Size / !D.Y_Size
```

It might take you a long time to figure out why your aspect ratio is always 0! The correct way to write this code is to force one of the integer values to be a float, like this:

```
aspect = Float(!D.X_Size) / !D.Y_Size
```



Now your *aspect* variable has the floating value you expect.

The other common way to get into trouble with integer variables is to not realize that integers in IDL are what are often called *short integers* in other programming languages. In other words, an integer in IDL is only two bytes long. Integers in most other programming languages are four bytes long (called a *long integer* in IDL).

A two-byte signed integer can only have positive values up to 32,767. Values greater than this “overflow” and are usually represented in IDL as negative numbers.

You can have trouble with short integers in two ways. First, you don’t take account of short integers in loops. For example, suppose you wanted to read a data file and you didn’t know how many lines there were in it. You might write a piece of code like this:

```
count = 0
WHILE NOT EOF(lun) DO BEGIN
```

Created for Frederick Walter

```

READF, lun, temp
data(count) = temp
count = count + 1
ENDWHILE

```

Data Type	Initialization	Index Generating
Byte	BytArr	BIndGen
16-Bit Signed Integer	IntArr	IndGen
32-Bit (Long) Signed Integer	LonArr	LIndGen
64-Bit Signed Integer	Lon64Arr	L64IndGen
16-Bit Unsigned Integer	UIntArr	UIndGen
32-Bit (Long) Unsigned Integer	ULonArr	ULIndGen
64-Bit Unsigned Integer	ULon64Arr	UL64IndGen
Floating Point	FltArr	FIndGen
Double Precision Floating	DblArr	DIndGen
Complex	ComplexArr	CIndGen
Double Precision Complex	DComplexArr	DCIndGen
String	StrArr	SIndGen
Pointer	PtrArr	None
Object	ObjArr	None

Table 2: *IDL functions that will create vectors and arrays of multiple dimensions, either initialized to 0 or initialized to their own index number.*

If you had more than 32,768 lines of data, this code would fail. The reason is that the variable *count* is initialized as an integer. It should be initialized as a long integer:

```

count = 0L
WHILE NOT EOF(lun) DO BEGIN
    READF, lun, temp
    data(count) = temp
    count = count + 1L
ENDWHILE

```

Now you can read as many lines as you like.

Another common place to find this error is in the counter for *For* loops. It is a good idea to always write your *For* loop command something like this:

```
FOR j=0L, num-1 DO ...
```

The second way you might get into trouble with short integers is when you try to read data that was produced by a program written in some other programming language (or vice versa). If you are going to read integer data produced by a C or Fortran program, you almost always want to be sure you are reading into *long* integers in IDL. Similarly, you will want to write long integer data to files that will be read by a C or Fortran program as integers.



Note that a new compiler option in IDL 5.3 can force IDL integers to be four-byte integers rather than two-byte integers by default. Normally this compiler option command is placed at the beginning of IDL procedures and functions. It causes the compiler to force all integer variables in that procedure or function to be four-byte long integers.

```
Compile_Opt DEFINT32
```

Working with Vectors and Arrays

IDL is a programming language that excels at working with data which is organized in vectors and arrays. (A model for the first version of IDL was APL, an excellent programming language for working with arrays.) To be an effective IDL programmer you must know how to perform mathematical operations on arrays. You will see many examples of how to do this in this book, but I wanted to call your attention to a couple of important points before you get started.

Creating Vectors

You can create a vector (a vector is just a one-dimensional array) or an array at the IDL command line by enclosing the vector values in square brackets, like this:

```
IDL> vector = [1, 2, 3]
```

This is an integer vector because the data values are integer values.

You can get information from IDL about the data type and organizational structure of variables by using the *Help* command, like this:

```
IDL> Help, vector
VECTOR           INT          = Array[3]
```

If you wanted to add a fourth element to this vector, this is easily done in IDL. Just type this, for example:

```
IDL> vector = [vector, 4]
IDL> Print, vector
1           2           3           4
```

Using Array Subscripts

Suppose you want to add another element between the second and third elements in the array. Then you can use array subscripting to help you do this. Array subscripts have their lower and upper bound separated by a colon. For example, you specify the first three elements of the vector above like this:

```
IDL> Print, vector(0:2)
1           2           3
```

Notice that vector subscripts start at 0 and not at 1. Notice also that vector subscripts use parentheses to distinguish themselves. This makes it difficult sometimes to distinguish a call to a function command from a subscripted array. To help with this problem, square bracket subscript notation was introduced in IDL 5.0. In other words, if you are running IDL 5.0 or higher you can type this:

```
IDL> Print, vector[0:2]
```

Beginning with IDL 5.3, you can force square bracket notation by setting a compiler option. Normally this compiler option command is placed at the beginning of IDL

procedures and functions. It causes the compiler to enforce square bracket subscript notation in that procedure or function.

```
Compile_Opt STRICTARR
```

This book uses square bracket subscripting to avoid any confusion with function calls. If you are using an IDL 4.x version of IDL, you will have to substitute parentheses for square brackets in the code to get the commands to work.

To use array subscripting to put another element between the second and third elements of the vector, you can do this:

```
IDL> vector = [vector[0:1], 5, vector[2:3]]  
IDL> Print, vector  
1 2 5 3 4
```

Vectors can also be created by using the array creation routines discussed above. For example, to create a 6-element floating-point vector with values ranging from 0 to 50, you can type this:

```
IDL> vector = FIndGen(6) * 10  
IDL> Print, vector  
0.000000 10.0000 20.0000 30.0000 40.0000 50.0000
```

Creating Arrays

Arrays can also be created from the IDL command line. For example, we can create a 3 column and 2 row array like this:

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

Your output in your IDL output window will look like this:

```
1 2 3  
4 5 6
```

Notice that this is identical to first creating a vector and then reformatting that vector into a 3 column by 2 row array with the *Reform* command, like this:

```
IDL> vector = IndGen(6) + 1  
IDL> array = Reform(vector, 3, 2)  
IDL> Print, array
```

What this tells you is that vectors and arrays are stored in IDL in *row* order. This becomes important when you are writing IDL programs because you will often want to take advantage of the way data is stored in IDL.

Accessing Elements in Arrays

Suppose you want to access the array element in the first column and second row of the array you just created. (The element with a value of 4.) You can do so like this:

```
IDL> Print, array[0,1]
```

Notice that the subscripts expect the column number and then the row number. This is just the opposite of what you might expect if you are used to working with matrices or arrays in linear algebra. (Notice also that the column and row number are one less than you really want. This is because array subscript indices start at 0, not 1.)

Column-row subscripting came about as a result of the astronomical image data IDL was originally written to handle. A row of data was a single scan line of an image. Storing the data in this fashion made data manipulation fast and accurate. Deciding whether a program uses column-row subscripting or row-column subscripting is

simply an arbitrary choice. There is no particular reason to choose one way or the other.

You can access the same element in this array using one-dimensional subscripts. Knowing that array elements are stored in row order, you want the fourth element in the array. You can access it like this:

```
IDL> Print, array[3]
```

The fact that you can access multi-dimensional arrays with one-dimensional subscripts is a powerful tool in many IDL programs.

You can also subscript arrays with vectors. For example, if you want to print the first, second, fourth, and sixth element of the array, you can type this:

```
IDL> indices = [0, 1, 3, 5]
IDL> Print, array[indices]
```

```
1          2          4          6
```

Extracting Vectors and Subarrays

IDL also makes it easy to extract vectors and subarrays from within arrays. For example, consider this data array, which is filled with random data:

```
IDL> data = RandomU(seed, 10, 20)
```

If you want to pull out the columns 6-10 and rows 12-15, you can type this:

```
IDL> subarray = data[5:9, 11:14]
```

If you want to plot, for example, just the 8th column of data, you can use the subscript symbol * to indicate all of the rows, like this:

```
IDL> Plot, data[7,*]
```

To create a vector of the 14th row, you can type:

```
IDL> vector = data[*,13]
```

To create an array of the last 5 rows of the data, type:

```
IDL> subarray = data[*, 15:19]
IDL> Help, subarray
```

You see that the subarray is now a 10 column by 5 row array.

You can also use the symbol * to mean “all the rest” of the data. For example, to create a subarray with the last 5 columns of the data, you can also type this:

```
IDL> subarray = data[5:*, *]
IDL> Help, subarray
```

You will learn more about arrays and how to work with them as you work through the examples in this book.

Working with IDL Graphics Windows

You will learn more about IDL graphics windows as you work through the examples in this book, but here are a couple of things it is good to know before you get started.

Creating Graphics Windows

First, a graphics window can be created directly with the *Window* command or indirectly by issuing a graphics display command when no other window is open. For example, you can create and open a window by typing this:

```
IDL> Window
```

Notice that the title bar of this window has a 0 in it. This is this window's *graphics window index number*. Each graphics window has an unique graphics window index number associated with it when the window is created. The *Window* command without any positional parameters always creates a window with window index number 0. We say this is "Window 0". You can have at least 128 graphics windows open at any one time in an IDL session. You can assign a graphics window index number for windows 0 through 31. IDL will assign graphics window index numbers for windows 32 through 127 by creating windows with the *Free* keyword (discussed below). For example, if you want to create a window with graphics window index number 10, you can type this:

```
IDL> Window, 10
```

If a window with the same graphics window index number already exists on the display, a *Window* command like this will first destroy the old one and then create a new one with this index number.

If you prefer (this is *always* a good idea when you are creating windows in IDL programs), you can open a window with a graphics window index number that is "free" or unused. The *Free* keyword is used for that purpose, like this:

```
IDL> Window, /Free
```

A window that is created with a *Free* keyword will have a graphics window index number greater than 31. The *Free* keyword is the only way to create a normal graphics window with an index number greater than 31.

Determining the Current Graphics Window

You now have at least three graphics windows open on the display. Only one of those windows is the *current graphics window*. The current graphics window is the window that will receive the output of a graphics command. The graphics window index number of the current graphics window is always stored in the *!D.Window* system variable. The value of *!D.Window* will be -1 if there are no graphics windows created and open to draw into.

This means, for example, that if you want to create a window and store its graphics window index number so you can later delete it or make it the active window (see details below), you can type something like this:

```
IDL> Window, /Free  
IDL> thisWindowIndex = !D.Window
```

Making a Graphics Window the Current Graphics Window

To make a window the current graphics window (so you can display graphics in it), you use the *WSet* command and the window's graphics index number. For example, suppose you want window 10 to be the current graphics window. You would type:

```
IDL> WSet, 10
```

All subsequent graphics commands will be displayed in window 10.



Note also that a window becomes the current graphics window when it is created. (This is not true of draw widget windows, however.) A window *must* be the current graphics window in order to draw graphics into it.

Deleting Graphics Windows

Graphics windows are deleted with the `WDelete` command and the window's graphics index number. A window does not have to be the current graphics window to be deleted. For example, to delete window 10, you can type this:

```
IDL> WDelete, 10
```

Here is a trick to delete all the graphics windows that exist on the display currently. Type:

```
IDL> WHILE !D.Window NE -1 DO WDelete, !D.Window
```

Positioning and Sizing Graphics Windows

Windows are positioned and sized according to an internal algorithm when they are created. You can position and size windows when you create them with keywords to the `Window` command. For example, to create a window that is 200 pixels wide and 300 pixels high, use the `XSize` and `YSize` keywords, like this:

```
IDL> Window, 1, XSize=200, YSize=300
```

Windows are positioned on the display with respect to the upper left-hand corner of the display in pixel or device coordinates. To position a window with its upper left-hand corner at location (75,150) on the display, use the `XPos` and `YPos` keywords, like this:

```
IDL> Window, 2, XPos=75, YPos=150
```

Bringing a Graphics Window Forward on the Display

Creating a graphics window gives that window the window focus and also makes it the current graphics window. That is, the graphics window is now the active window with respect to the Window Manager. (Just because a graphics window has the window focus does not mean, in general, that it is the current graphics window.) To type a command, you have to move the window focus back to the command window. On some platforms, especially PCs, this causes the graphics window to pop back behind other windows.

Or sometimes a graphics window just gets behind other windows on the display and you would like to bring it forward so you can see it. To bring a window forward on the display, without changing the window focus, use the `WShow` command and the window's graphics index number, like this:

```
IDL> WShow, 1
```

Notice that your cursor and the window focus are still in the command window or window where you are typing IDL commands.

Bringing a window forward with the `WShow` command does *not* make the window the current graphics window. If you want to move a window forward and make it the current graphics window (assuming it is not the current window already), then you have to type both a `WShow` and a `WSet` command, like this:

```
IDL> WShow, 2
IDL> WSet, 2
```



Note that typing `WShow` without a parameter will bring the current graphics window forward on the display. This is especially helpful when the current graphics window is obscured and you want to move it forward on the display without removing the focus from the IDL command window.

```
IDL> WShow
```



Note that on PCs and Macintosh computers, you can use the ALT-TAB or OPTION-TAB keys, respectively, to cycle through and select windows that are open but not currently visible on the display and make them the window with the window focus.

Putting a Title on a Graphics Window

Sometimes you would like your graphics window to have a more descriptive title than just its graphics window index number. You can use the *Title* keyword to put a title on the window, like this:

```
IDL> Window, Title='Example IDL Graphics Commands'
```

Erasing a Graphics Window

To erase the current graphics window, you can use the *Erase* command, like this:

```
IDL> Erase
```

If you want to erase the current graphics display with a particular color index (or 24-bit color value, if you are on a 24-bit display), you can use the *Color* keyword. For example, you can erase the current graphics display with a charcoal gray color by typing this:

```
IDL> Device, Decomposed=0  
IDL> TVLCT, 70, 70, 70, 100  
IDL> Erase, Color=100
```

To erase graphics windows that are *not* the current graphics window (i.e., the window indicated by the system variable *!D.Window*), you must first make the window the current graphics window and then issue the *Erase* command.

Chapter 2

♦ Discovering the Possibilities ♦♦♦



Simple Graphical Displays

Chapter Overview

The bread and butter of scientific analysis is the ability to see your data as simple line plots, contour plots and surface plots. In this chapter, you will learn how easy it is to display your data this way. You will also learn to use system variables and keywords to position and annotate simple graphical displays.

Specifically, you will learn:

- How to display data as a line plot with the *Plot* command
- How to display data as a surface with the *Surface* and *Shade_Surf* commands
- How to display data as a contour plot with the *Contour* command
- How to position simple graphical displays in the display window
- How to use common keywords to annotate and customize your graphical displays

Simple Graphical Displays in IDL

A simple graphical display in IDL is an example of what is called a *raster graphic*. That is to say, using a *Plot*, or *Contour*, or *Surface* command generates an algorithm that lights up certain pixels in the display window. Such raster graphics have no persistence. In other words, once IDL has displayed the graphic and lighted up certain pixels it has no knowledge of what it has done. This means, for example, that IDL has no way to respond to the user resizing the display window. The graphics display cannot, in general, be redrawn in such circumstances except by re-issuing the graphics command over again.

Nevertheless, raster graphics commands are widely used in IDL because they are fast and simple to use. And, as you will see, many of the limitations often associated with raster graphic commands can be overcome if you are careful in how you write IDL programs that use raster graphic commands. This chapter introduces you to the concepts you will need to know to write resizable IDL graphics windows or produce hardcopy output directly with raster graphic commands. The graphics commands in this chapter are concerned with what Research Systems calls *direct graphics*.

A different type of graphics option—named *object graphics* by Research Systems—was introduced in IDL 5.0. Object graphics commands are considerably harder to use, but they also give you more power and flexibility in how your IDL graphic programs

are written. Object graphics commands are not really meant to be typed at the IDL command line. Rather, they are included in IDL programs, especially widget programs (programs with graphical user interfaces). Object graphics commands are discussed later in this book.

Creating Line Plots

The simplest way to generate a line plot is to plot a vector. You can use the *LoadData* command to open the *Time Series Data* data set. The *LoadData* command is one of the IDL programs that came with this book. (See “IDL Programs and Data Files Used in the Book” on page 4 for more information.) It is used to load the data sets used in the programming examples in this book. To see the data sets available to you, type this:

```
IDL> curve = LoadData()
```



If you forgot to include the parentheses when you issued the *LoadData* command above, you may need to re-compile the *LoadData* program before it will work properly. The reason for this is that IDL now thinks “*loaddata*” is a variable and acts accordingly when it sees it in a command line. Re-compiling the function will get the “*loaddata*” name on IDL’s list of function names where it belongs. Type this:

```
IDL> .Compile LoadData
```

The *Time Series Data* data set is the first data set on the *LoadData* list. Click on it. The data is now loaded into the variable *curve*. Another way to select this first data set is to call *LoadData* like this:

```
IDL> curve = LoadData(1)
```

To see how the variable *curve* is defined, type this:

```
IDL> Help, curve
CURVE           FLOAT      = Array[101]
```

You see that *curve* is a floating point vector (or one-dimensional array) of 101 elements.

To plot the vector, type:

```
IDL> Plot, curve
```

IDL will try to plot as nice a plot as it possibly can with as little information as it has. In this case, the X or horizontal axis is labeled from 0 to 100, corresponding to the number of elements in the vector, and the Y or vertical axis is labeled with data coordinates (i.e., this is the dependent data axis).

But most of the time a line plot displays one data set (the independent data) plotted against a second data set (the dependent data). For example, the curve above may represent a signal that was collected over some period of time. You might want to plot the value of the signal at some moment of time. In that case, you need a vector the same length as the curve vector (so you can have a one-to-one correspondence) and scaled into the units of time for the experiment. For example, you can create a time vector and plot it against the curve vector, like this:

```
IDL> time = FIndGen(101)*(6.0/100)
IDL> Plot, time, curve
```

The *FIndGen* command creates a vector of 101 elements going from 0 to 100. The multiplication factor scales each element so that the final result is a vector of 101 elements going from 0 to 6. Your graphical output should look similar to that in Figure 1.

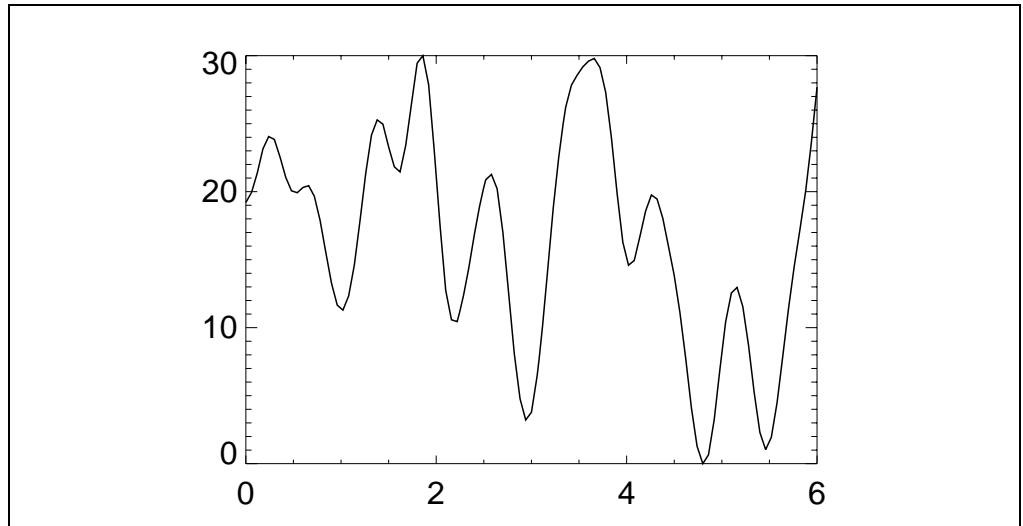


Figure 1: A plot of the independent data (time) versus the dependent data (curve).

Notice that there are no titles on the axes associated with this plot. Placing titles on graphics plots is easy. Just use the *XTitle* and *YTitle* keywords. For example, to label the curve plot, you can type this:

```
IDL> Plot, time, curve, XTitle='Time Axis', $  
      YTitle='Signal Strength'
```

You can even put a title on the entire plot by using the *Title* keyword, like this:

```
IDL> Plot, time, curve, XTitle='Time Axis', $  
      YTitle='Signal Strength', Title='Experiment 35M'
```

Your output will look similar to the illustration in Figure 2. Note that the display shows white lines on a black background, while the illustration shows black lines on a white background. These illustrations are encapsulated PostScript files produced by IDL. It is common for the drawing and background colors to be reversed in PostScript files. (See “Problem: PostScript Devices Use Background and Plotting Colors Differently” on page 189 for more information.)

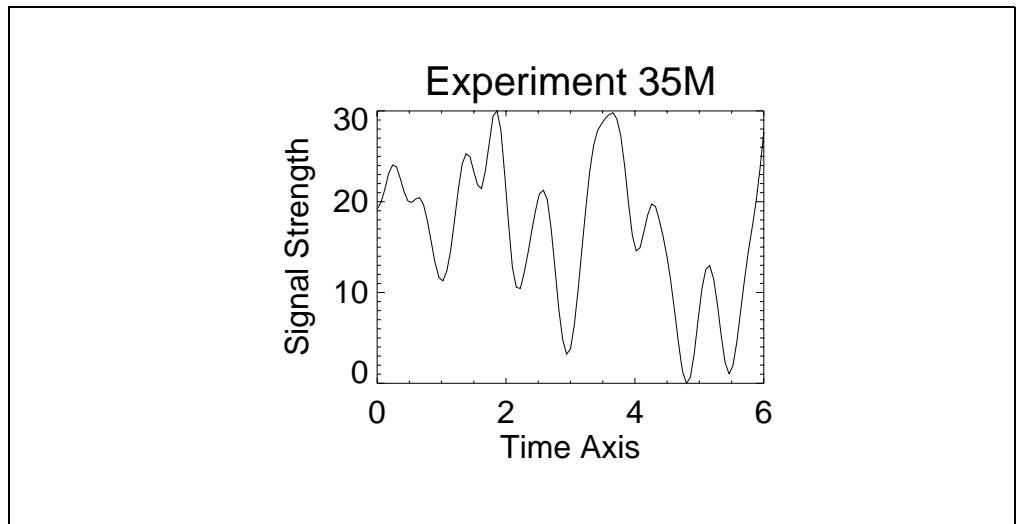


Figure 2: A simple line plot with axes labels and a plot title.

Notice that the plot title is slightly larger than the axes labels. In fact, it is 1.25 times as large. You can change the size of all the plot annotations with the *CharSize* keyword. For example, if your eyes are like mine, you might want to make the axis characters about 50% larger, like this:

```
IDL> Plot, time, curve, XTitle='Time Axis', $  
      YTitle='Signal Strength', Title='Experiment 35M', $  
      CharSize=1.5
```

If you want the character size on all your graphic displays to be larger than normal, you can set the *CharSize* field on the plotting system variable, like this:

```
IDL> !P.CharSize = 1.5
```

Now all subsequent graphics plots will have larger characters, unless specifically overruled by the *CharSize* keyword on a graphics output command.

You can even change the character size of each individual axis by using the *[XYZ]CharSize* keyword. For example, if you want the Y axis annotation to be twice the size of the X axis annotation, you can type this:

```
IDL> Plot, time, curve, XTitle='Time Axis', XCharSize=1.0, $  
      YTitle='Signal Strength', YCharSize=2.0
```



Note that the *[XYZ]CharSize* keywords use the current character size as the base from which they calculate their own sizes. The current character size is always stored in the system variable *!P.CharSize*. This means that if you set the *XCharSize* keyword to 2 when the *!P.CharSize* system variable is also set to 2, then the characters will be four times their normal size.

Customizing Graphics Plots

This is a simple line plot, without much information associated with it, aside from the data itself. But there are many ways to customize and annotate line plots. The *Plot* command can be modified with over 50 different keywords. Here are a few of the things you might want to do:

- modify line styles or thicknesses
- use symbols with or without lines between them
- create your own plotting symbols
- add color in your plot to highlight important features
- change the length of tick marks or the number of tick intervals
- use logarithmic scaling on the plot axes
- change the data range to plot just the subset of the data you are interested in
- eliminate axes or otherwise change the style of the plot

Modifying Line Styles and Thicknesses

Suppose, for example, you wanted to plot your data with different line styles. For a line with a long dash line style, you can try this:

```
IDL> Plot, time, curve, LineStyle=5
```

Different line styles are selected for line plots by using the *LineStyle* keyword with one of the index numbers shown in Table 3. For example, if you wished to plot the curve with dashed line, you set the *LineStyle* keyword to a value of 2, like this:

```
IDL> Plot, time, curve, LineStyle=2
```

Index Number	Line Style
0	Solid
1	Dotted
2	Dashed
3	Dash Dot
4	Dash Dot Dot
5	Long Dash

Table 3: *The line style can be changed by assigning these index numbers to the `LineStyle` keyword.*

The thickness of lines used on line plots can also be changed. For example, if you wanted the plot displayed with dashed lines that were three times thicker than normal you can type this:

```
IDL> Plot, time, curve, LineStyle=2, Thick=3
```

Displaying Data with Symbols Instead of Lines

Suppose you wanted to plot your data with symbols instead of lines. Like the `LineStyle` keyword, similar index numbers exist to allow you to choose different symbols for your line plots. Table 4 shows you the possible index numbers you can use with the `PSym` (*Plotting Symbol*) keyword. For example, you can draw the plot with asterisks by setting the `PSym` keyword to 2, like this:

```
IDL> Plot, time, curve, PSym=2
```

Your output should look similar to the output in Figure 3.

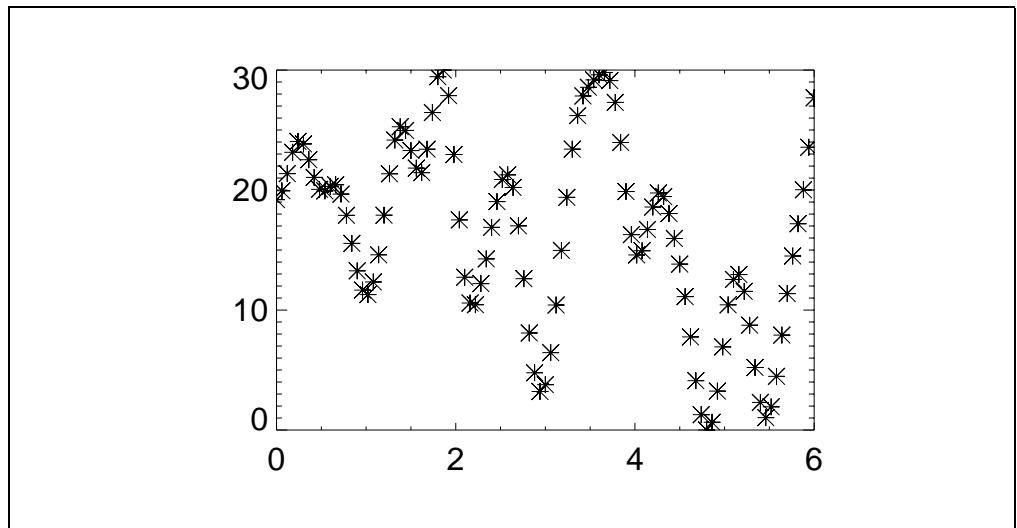


Figure 3: *A line plot with the data plotted as symbols instead of as a line.*

Displaying Data with Lines and Symbols

You can connect your plot symbols with lines by using a negative value for the `PSym` keyword. For example, to plot your data with triangles connected by a solid line, type

```
IDL> Plot, time, curve, PSym=-5
```

To create larger symbols, add the *SymSize* keyword like this. Here the symbol is twice the “normal” size. A value of 4 would make the symbol four times its normal size, and so on.

```
IDL> Plot, time, curve, PSym=-5, SymSize=2.0
```

Index Number	Symbol Drawn on Plot
0	No symbol is used and points are connected by lines.
1	Plus sign
2	Asterisk
3	Period
4	Diamond
5	Triangle
6	Square
7	X
8	User defined (with the <i>UserSym</i> procedure).
9	This index is not used for anything!
10	Data is plotted in histogram mode.
-PSym	Negative values of <i>PSym</i> connect symbols with lines.

Table 4: *These are the index numbers you can use with the *PSym* keyword to produce different plotting symbols on your plots. Note that using a negative number for the plotting symbol will connect the symbols with lines.*

Creating Your Own Plotting Symbols

If you feel creative, you can even create your own plotting symbols. The *UserSym* command is used for this purpose. After you have created your special symbol, you select it by setting the *PSym* keyword equal to 8. Here is an example that creates an star as a symbol. The vectors *x* and *y* define the vertices of the star as offsets from the origin at (0,0). You can create a filled symbol by setting the *Fill* keyword on the *UserSym* command.

```
IDL> x = [0.0, 0.5, -0.8, 0.8, -0.5, 0.0]
IDL> y = [1.0, -0.8, 0.3, 0.3, -0.8, 1.0]
IDL> TvLCT, 255, 255, 0, 150
IDL> UserSym, x, y, Color=150, /Fill
IDL> Device, Decomposed=0
IDL> Plot, time, curve, PSym=-8, SymSize=2.0
```

Your output should look similar to the output in Figure 4.

Drawing Line Plots in Color

You can draw your line plots in different colors. (Colors are discussed in detail in “Working with Colors in IDL” on page 81. For now, just type the *TvLCT* command below. You will learn exactly what the command means later. Basically, you are loading three color vectors that describe the red, green, and blue components of the

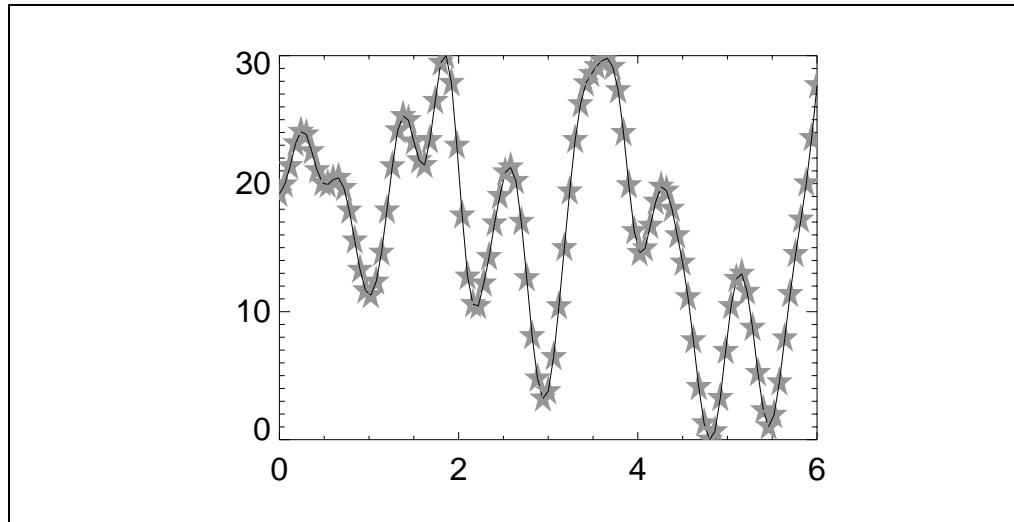


Figure 4: A plot with symbols created by the `UserSym` procedure.

color triples that describe the charcoal, yellow, and green colors.) For example, load a charcoal, yellow and green color into color indices 1, 2, and 3 like this:

```
IDL> TVLCT, [70, 255, 0], [70, 255, 255], [70, 0, 0], 1
```

To draw the plot in yellow on a charcoal background, type:

```
IDL> Plot, time, curve, Color=2, Background=1
```



If nothing appears in your display window when you type the command above, it is likely because you are running IDL on a 24-bit color display and you have color decomposition turned on. You will learn more about color decomposition later, but for now, be sure color decomposition is off. Type this to produce the proper colors:

```
IDL> Device, Decomposed=0
IDL> Plot, time, curve, Color=2, Background=1
```

If you want just the line to be a different color, you must first plot the data with the `NoData` keyword turned on, then overplot the line with the `OPlot` command (discussed below). For example, to have a yellow plot on a charcoal background, with the data in green, type:

```
IDL> Plot, time, curve, Color=2, Background=1, /NoData
IDL> OPlot, time, curve, Color=3
```

Limiting the Range of Line Plots

Not all of your data must be plotted on a line plot. You can limit the amount of data you plot with keywords. For example, to plot just the data that falls between 2 and 4 on the X axis, type:

```
IDL> Plot, time, curve, XRange=[2, 4]
```

Or, to plot just the portion of the data that has Y values between 10 and 20 and X values that fall between 2 and 4, type:

```
IDL> Plot, time, curve, YRange=[10, 20], XRange=[2, 4]
```

You can reverse the direction of the data by specifying the data range with keywords. For example, to draw the plot with the Y axis reversed, type this:

```
IDL> Plot, time, curve, YRange=[30, 0]
```

Your output should look similar to the illustration in Figure 5, below.

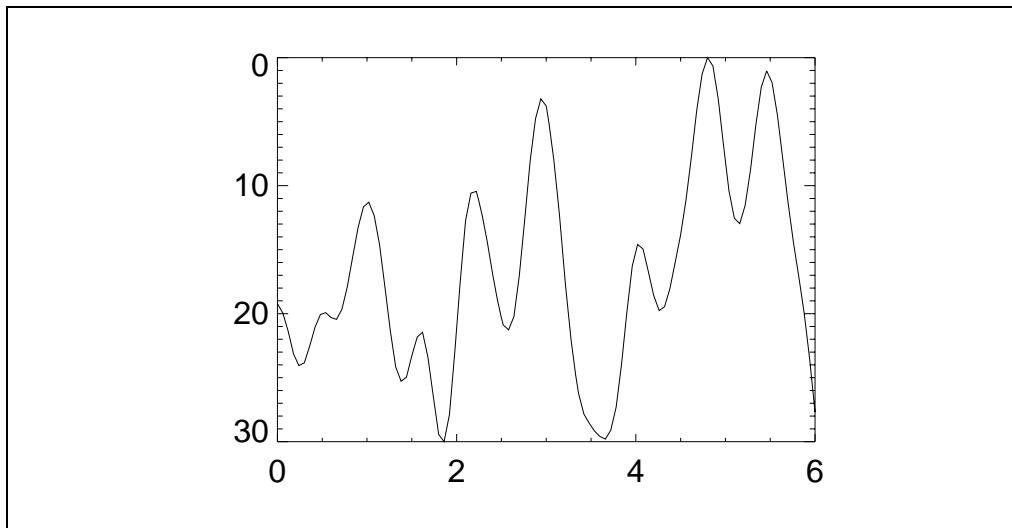


Figure 5: A plot with the zero point of the Y axis located on the top of the plot.



If your chosen axis range doesn't fall within IDL's sense of what an aesthetically pleasing axis range should be, IDL may ignore the asked-for range. For example, try this command:

```
IDL> Plot, time, curve, XRange=[2.45, 5.64]
```

The X axis range goes from 2 to 6, which is not exactly what you asked IDL to do. To make sure you always get the axis range you ask for, set the *XStyle* keyword to 1, like this:

```
IDL> Plot, time, curve, XRange=[2.45, 5.64], XStyle=1
```

You will learn more about the *[XYZ]Style* keywords in the next section.

Changing the Style of Line Plots

It is possible to change many characteristics of your plots, including the way they look. For example, you may not care for the box style of the line plots. If not, you can change the plot characteristics with the *[XYZ]Style* keywords. The values you can use with these keywords to change line plot styles is shown in Table 5. For example, to eliminate box axes and have just one X axis and one Y axis, type:

```
IDL> Plot, time, curve, XStyle=8, YStyle=8
```

Value	Description of Axis Property Affected
1	Force exact axis range.
2	Extend axis range.
4	Suppress entire axis.
8	Suppress box style axis. (Draw only one axis.)
16	Inhibit setting the Y axis starting value to 0. (Y axis only.)

Table 5: A table of the keyword values for the *[XYZ]Style* keywords that set properties for the axis. Note that values can be added to specify more than one axis property.

You can turn an axis off completely. For example, to show the plot with a single Y axis, you can type:

```
IDL> Plot, time, curve, XStyle=4, YStyle=8
```

Your output should look similar to the illustration in Figure 6.

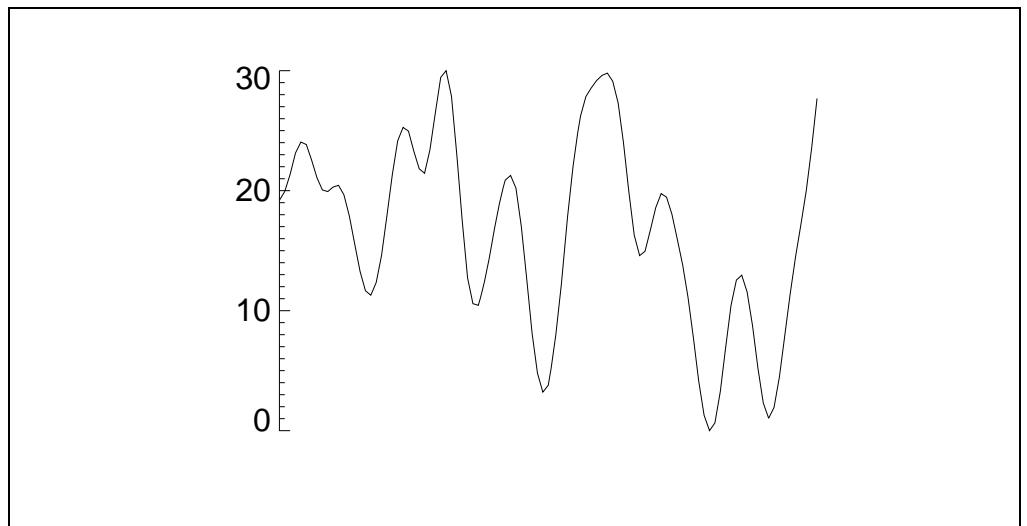


Figure 6: A plot with the X axis suppressed and Y box axes turned off.

You could show the same plot with Y axes and Y grid lines:

```
IDL> Plot, time, curve, XStyle=4, YTICKLen=1, YGridStyle=1
```

The *[XYZ]Style* keyword can be used to set more than one axis property at a time. This is done by adding the appropriate values together. For example, you see from Table 5 that the value to force the exact axis range is 1, and the value to suppress the drawing of box axes is 8. To both make an exact X axis range and to suppress box axes, these values are added together, like this:

```
IDL> Plot, time, curve, XStyle=8+1, XRange=[2,5]
```

To create a full grid on a line plot is accomplished (strangely) with the *TickLen* keyword, like this:

```
IDL> Plot, time, curve, TickLen=1
```

Outward facing tick marks are created with a negative value to the *[XYZ]TickLen* keywords. For example, to create all outward facing tick marks, type:

```
IDL> Plot, time, curve, TickLen=-0.03
```

To create outward facing tick marks on individual axes, use the *[XYZ]TickLen* keywords. For example, to have outward facing tick marks on just the X axis, type:

```
IDL> Plot, time, curve, XTickLen=-0.03
```

You can also select the number of major and minor tick marks on an axis with the *[XYZ]Ticks* and *[XYZ]Minor* keywords. For example to create the plot with only two major tick intervals, each with 10 minor tick marks, on the X axis, type:

```
IDL> Plot, time, curve, XTicks=2, XMinor=10, XStyle=1
```

Plotting Multiple Data Sets on Line Plots



You are not restricted to just a single data set on a line plot. IDL allows you to plot as many data sets as you like on the same set of axes. The *OPlot* command is used for this purpose. Type the following commands. Your output will look like the illustration in Figure 7.

```
IDL> Plot, curve
IDL> OPlot, curve/2.0, LineStyle=1
IDL> OPlot, curve/5.0, LineStyle=2
```

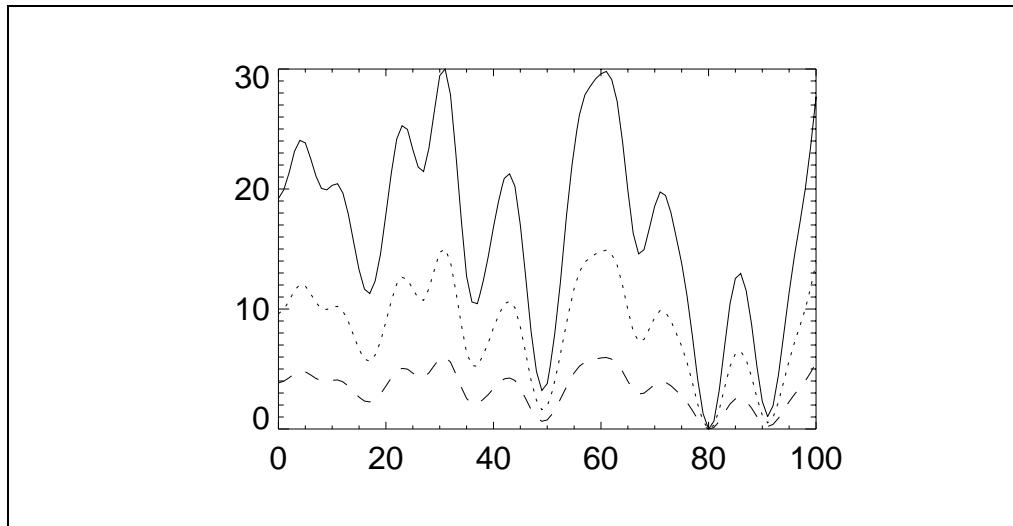


Figure 7: An unlimited number of data sets can be plotted on the same line plot.

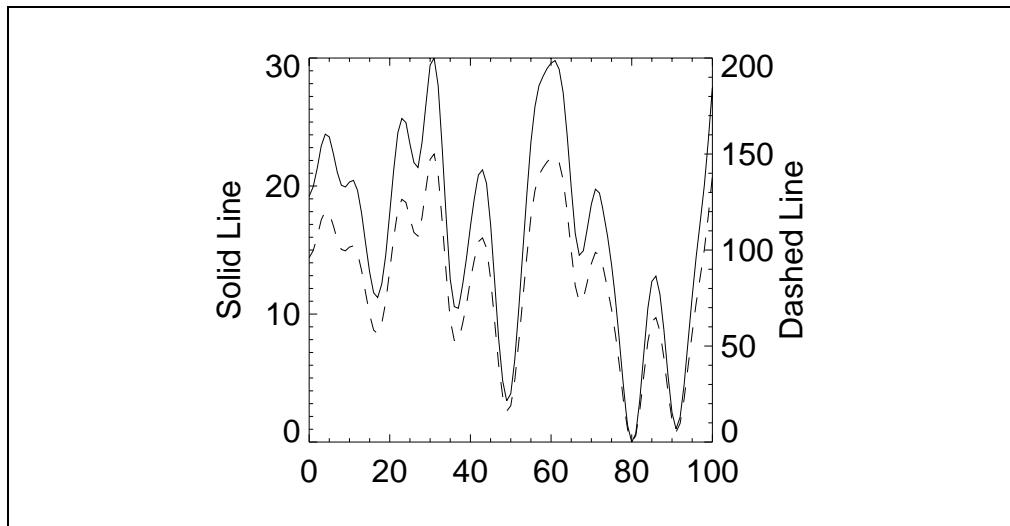


Figure 8: A line plot with two Y axes. The second axis is positioned with the *Axis* command. Be sure to save the data scaling with the *Save* keyword.



The initial *Plot* command establishes the data scaling (in the *!X.S* and *!Y.S* scaling parameters) for all the subsequent plots. In other words, it is the values in the *!X.S* and *!Y.S* system variable that tells IDL how to take a point in data space and place it on the display in device coordinate space. Make sure the initial plot has axis ranges sufficient to encompass all subsequent plots, or the data will be clipped. Use the *XRange* and *YRange* keywords in the first *Plot* command to create a data range large enough. To distinguish different data sets, you can use different line styles, different colors, differ-

ent plot symbols, etc. The *OPlot* command accepts many of the same keywords the *Plot* command accepts.

```
IDL> TvLCT, [255, 255, 0], [0, 255, 255], [0, 0, 0], 1
IDL> Plot, curve, /NoData
IDL> OPlot, curve, Color=1
IDL> OPlot, curve/2.0, Color=2
IDL> OPlot, curve/5.0, Color=3
```

Plotting Data on Multiple Axes

Sometimes you wish to have two or more data sets on the same line plot, but you want the data sets to use different Y axes. It is easy to establish as many axes as you need with the *Axis* command. The key to using the *Axis* command is to use the *Save* keyword to save the proper plot scaling parameters (i.e., those stored in the *!X.S* and *!Y.S* system variables) for subsequent plotting calls.

For example, here one plot is drawn and the *Axis* command along with the *Save* keyword is used to establish a second Y axis. The curve in the *OPlot* command will use the scaling factors saved by the *Axis* command to determine its placement on the plot. The proper commands are these:

```
IDL> Plot, curve, YStyle=8, YTitle='Solid Line', $
      Position = [0.15, 0.15, 0.85, 0.95]
IDL> Axis, YAxis=1, YRange=[0,200], /Save, $
      YTitle='Dashed Line'
IDL> OPlot, curve*5, LineStyle=2
```

The *Position* keyword is used to position the first plot on the page. To learn more about the *Position* keyword, see “Positioning Graphic Output in the Display Window” on page 44. Your output will look like the illustration in Figure 8.

Creating Surface Plots

Any two-dimensional data set can be displayed as a surface (with automatic hidden-line removal) in IDL with a single *Surface* command. First, you must open a data file. Use the *LoadData* command to open the *Elevation Data* data set, like this:

```
IDL> peak = LoadData(2)
```

You can see that this is a 41-by-41 floating point array by typing the *Help* command, like this:

```
IDL> Help, peak
```

This data can be viewed as a surface with a single command:

```
IDL> Surface, peak, CharSize=1.5
```

Your output should look similar to the illustration in Figure 9.

Notice that if the *Surface* command is given just a single array as an argument it plots the array as a function of the number of elements (41 in the X and Y directions, in this case) in the array. (You used the *CharSize* keyword to increase the character size so you can see this easily.) But, as you did before with the *Plot* command, you can specify values for the X and Y axes that may have some physical meaning for you. For example, the X and Y values may be longitude and latitude coordinates. Here we make a latitude vector that extends from 24 to 48 degrees of latitude and a longitude vector that extends from -122 to -72 degrees of longitude:

```
IDL> lat = FIndGen(41) * (24.0/40) + 24
IDL> lon = FindGen(41) * (50.0/40) - 122
```

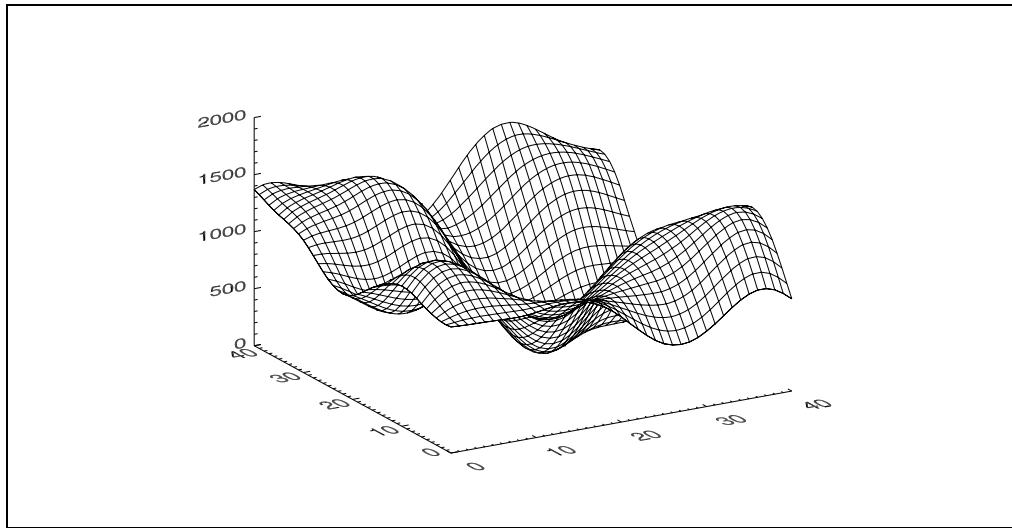


Figure 9: A basic surface plot of elevation data.

```
IDL> Surface, peak, lon, lat, XTitle='Longitude', $  
      YTitle='Latitude', ZTitle='Elevation', CharSize=1.5
```

Your output will look like the illustration in Figure 10.

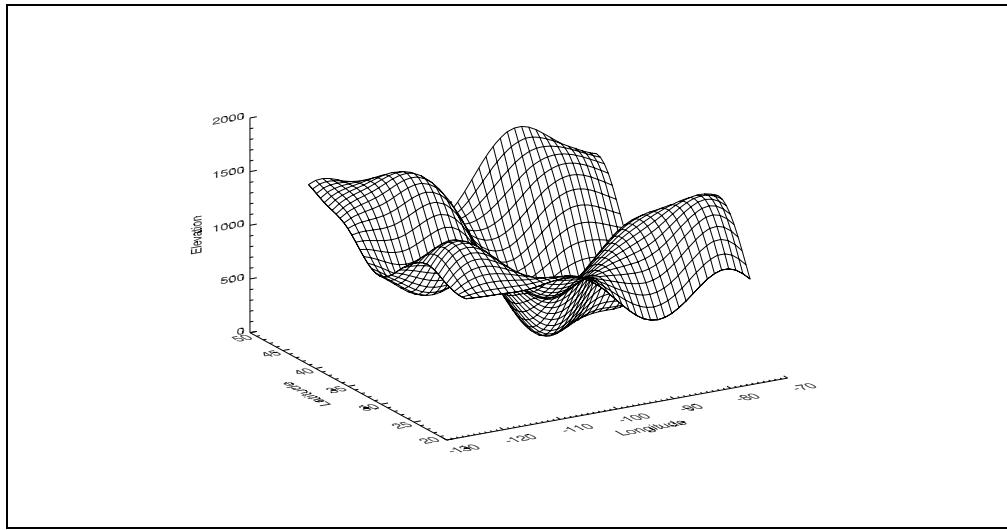


Figure 10: A surface plot with meaningful values associated with its axes.

The parameters *lon* and *lat* in the command above are monotonically increasing and regular. They describe the locations that connect the surface grid lines. But the grid does not have to be regular. Consider what happens if you make the longitudinal data points irregularly spaced. For example, you can simulate randomly spaced longitudinal points by typing this:

```
IDL> seed = -1L  
IDL> newlon = RandomU(seed, 41) * 41  
IDL> newlon = newlon[Sort(newlon)] * (24./40) + 24  
IDL> Surface, peak, newlon, lat, XTitle='Longitude', $  
      YTitle='Latitude', ZTitle='Elevation', CharSize=1.5
```

You see now that the longitudinal X values are not regularly spaced. Although it may look like the data is being re-sampled, it is not. You are simply drawing the surface

grid lines at the locations specified by the longitude and latitude data points. Your output should be similar to the illustration in Figure 11.

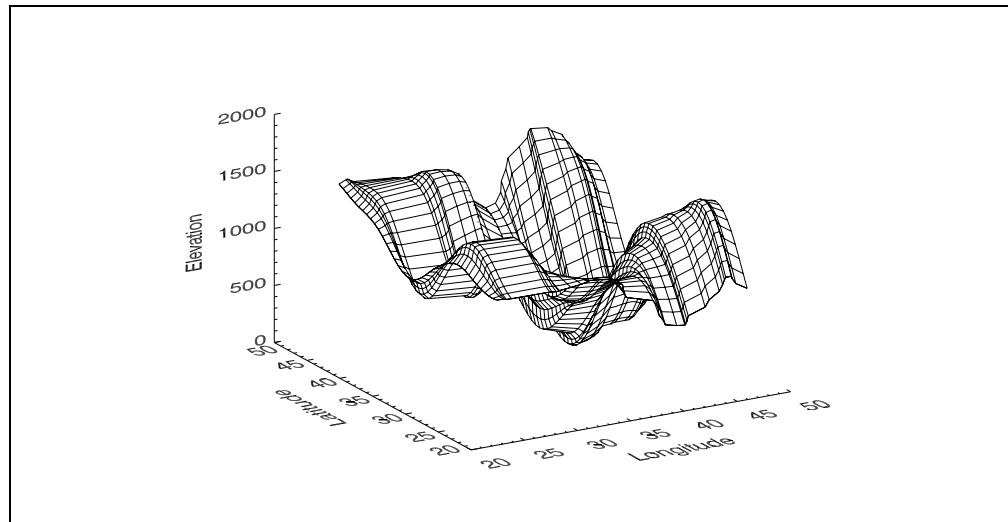


Figure 11: The same surface plot, but with an irregularly spaced X vector.

Customizing Surface Plots

There are over 70 different keywords that can modify the appearance of a surface plot. In fact, many of these keywords you already learned for the *Plot* command. For example, you saw in the code above that the same title keywords can be used to annotate the axes. Notice, however, that when you add a *Title* keyword that the title is rotated so that it is always in the XZ plane of the surface plot. For example, type this:

```
IDL> Surface, peak, lon, lat, XTitle='Longitude', $  
      YTitle='Latitude', Title='Mt. Elbert', Charsize=1.5
```

This is not always what you want. If you want the plot title to be in a plane parallel to the display, you will have to draw the surface with the *Surface* command, and the title with the *XYOutS* command. (There is detailed information on the *XYOutS* command on page 51.) For example, like this:

```
IDL> Surface, peak, lon, lat, XTitle='Longitude', $  
      YTitle='Latitude', Charsize=1.5  
IDL> XYOutS, 0.5, 0.90, /Normal, Size=2.0, Align=0.5, $  
      'Mt. Elbert'
```

Rotating Surface Plots

You may want to rotate the angle at which you view a surface plot. A surface plot can be rotated about the X axis with the *Ax* keyword and about the Z axis with the *Az* keyword. Rotations are expressed in degrees counterclockwise as you look from positive values on the axis towards the origin. The *Az* and *Ax* keywords default to 30 degrees if they are omitted. For example, to rotate the surface about the Z axis by 60 degrees and about the X axis by 35 degrees, type:

```
IDL> Surface, peak, lon, lat, Az=60, Ax=35, Charsize=1.5
```

Your output should look like the illustration in Figure 12.

Although direct graphics surface commands are still useful in IDL, many programmers are finding that surfaces rendered with the object graphics class library

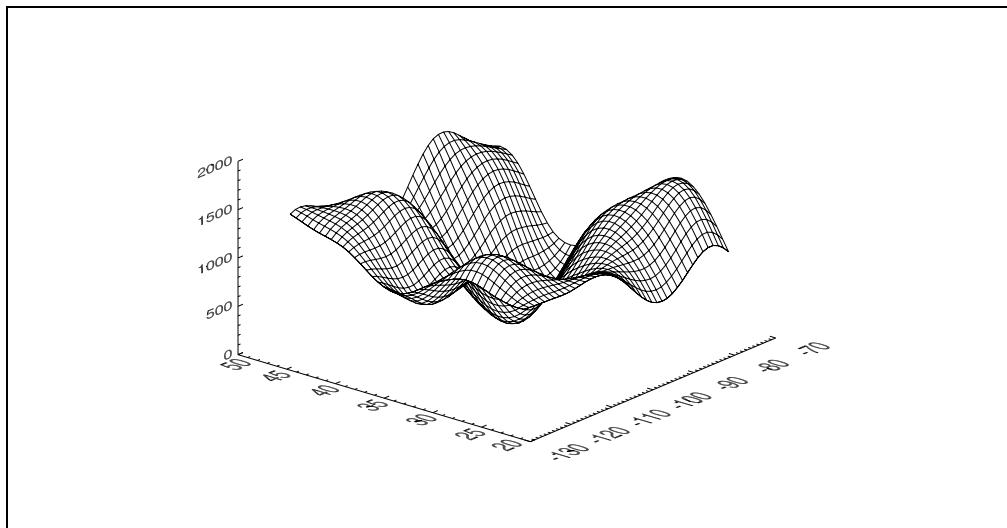


Figure 12: Surface plots can be rotated with the *Ax* and *Az* keywords.

are much more useful to them, especially when surface rotations are desired. Partly this is because direct graphics surface rotations have an artificial limitation that requires the Z axis to be pointing in a vertical direction in the display window. No such limitation exists for object graphics surfaces.

To see what some of the advantages are, run the program *FSC_Surface*, which is among the program files you downloaded to use with this book. Press and drag the cursor in the graphics window to rotate the surface plot. Controls in the menu bar give you the opportunity to set many properties of the surface plot, including shading parameters, lights, and colors. The *FSC_Surface* program looks similar to the illustration in Figure 13.

Adding Color to a Surface Plot

Sometimes you want to add color to the surface plot to emphasize certain features. It is easy to add color, using many of the same keywords that you used to add color to line plots. (Colors are discussed in detail in “Working with Colors in IDL” on page 81. For now, just type the *TvLCT* command below. You will learn exactly what the command means later. Basically, you are loading three color vectors that describe the red, green, and blue components of the color triples that describe the charcoal, yellow, and green colors.) For example, to create a yellow surface plot on a charcoal background, type:

```
IDL> TvLCT, [70, 255, 0], [70, 255, 255], [70, 0, 0], 1
IDL> Device, Decomposed=0
IDL> Surface, peak, Color=2, Background=1
```

If you wanted the underpart of the surface to be a different color than the top, say green, you can use the *Bottom* keyword like this:

```
IDL> Surface, peak, Color=2, Background=1, Bottom=3
```

If you wanted the axes to be drawn in a different color—green, for example—than the surface, you have to enter two commands. The first command uses the *NoData* keyword so that just the axes are drawn, while the second command draws the surface itself with the axes turned off. See Table 5 on page 26 for a list of values you can use with the */XYZ/Style* keywords and what they mean.

```
IDL> Surface, peak, Color=3, /NoData
IDL> Surface, peak, /NoErase, Color=2, Bottom=1, $
      XStyle=4, YStyle=4, ZStyle=4
```

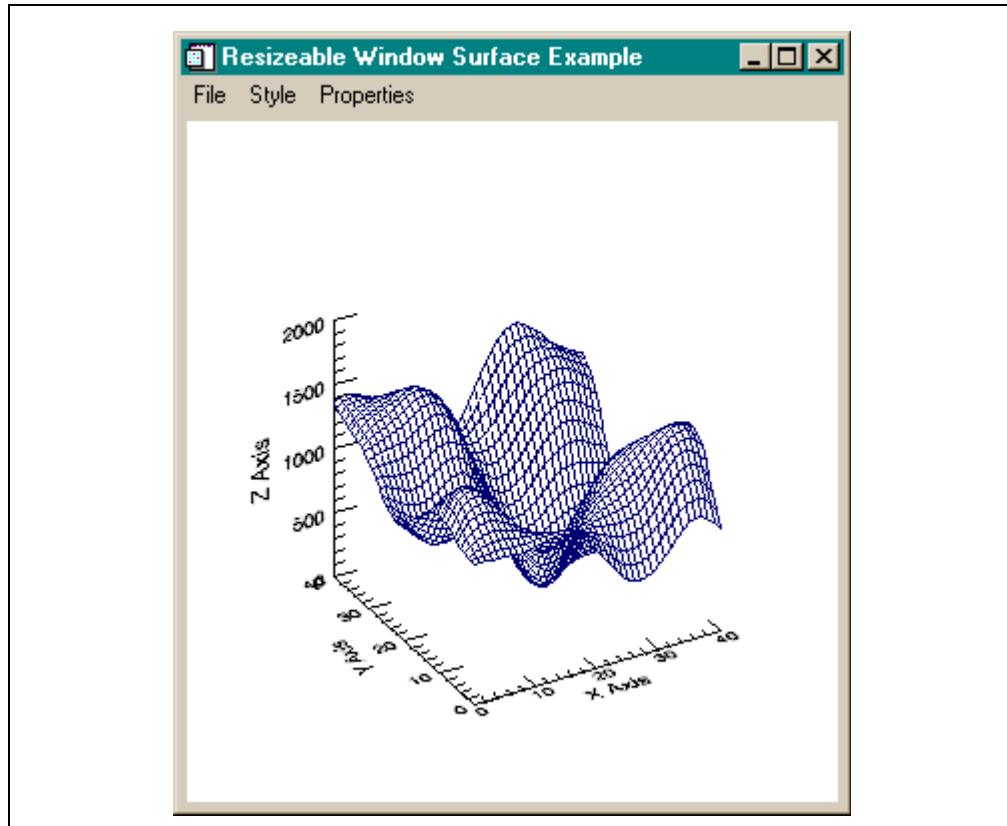


Figure 13: The FSC_Surface program. Click and drag in the graphics window to rotate this surface plot. The program is written using the object graphics class library. Properties of the surface can be changed via options in the menu bar.

It is also possible to draw the mesh lines of the surface in different colors representing a different data parameter. For example, you can think of draping a second data set over the first, coloring the mesh of the first data set with the information contained in the second.

To see how this would work, open a data set named *Snow Pack* and display it as a surface with these commands. Notice that the *Snow Pack* data set is the same size as the *peak* data set, a 41-by-41 floating point array.

```
IDL> snow = LoadData(3)
IDL> Help, snow
IDL> Surface, snow
```

Now, you will drape the data in the variable *snow* over the top of the data in the variable *peak*, using the values of *snow* to color the mesh of *peak*. First, load some colors in the color table with the *LoadCT* command. The actual shading is done with the *Shades* keyword, like this:

```
IDL> LoadCT, 5
IDL> Surface, peak, Shades=BytScl(snow, Top=!D.Table_Size-1)
```



Notice you had to scale the *snow* data set (with the *BytScl* command) into either the number of colors you are using in this IDL session or the size of the color table. If you failed to scale the data, you might see a set of axes and no surface display at all. This is because the data must be scaled into the range 0 to 255 to be used as the shading parameters for the surface.

Sometimes you might like to display the surface colored according to elevation. This is easily done just by using the data itself as the shading parameter.

```
IDL> Surface, peak, Shades=BytScl(peak, Top=!D.Table_Size-1)
```

Modifying the Appearance of a Surface Plot

There are a number of keywords you can use to modify the appearance or style of a surface plot. For example, you can display the surface with a skirt. Use the *Skirt* keyword to indicate the value where the skirt should be drawn. For example, try these commands:

```
IDL> Surface, peak, Skirt=0  
IDL> Surface, peak, Skirt=500, Az=60
```

The output of the first command above should look similar to the illustration in Figure 14.

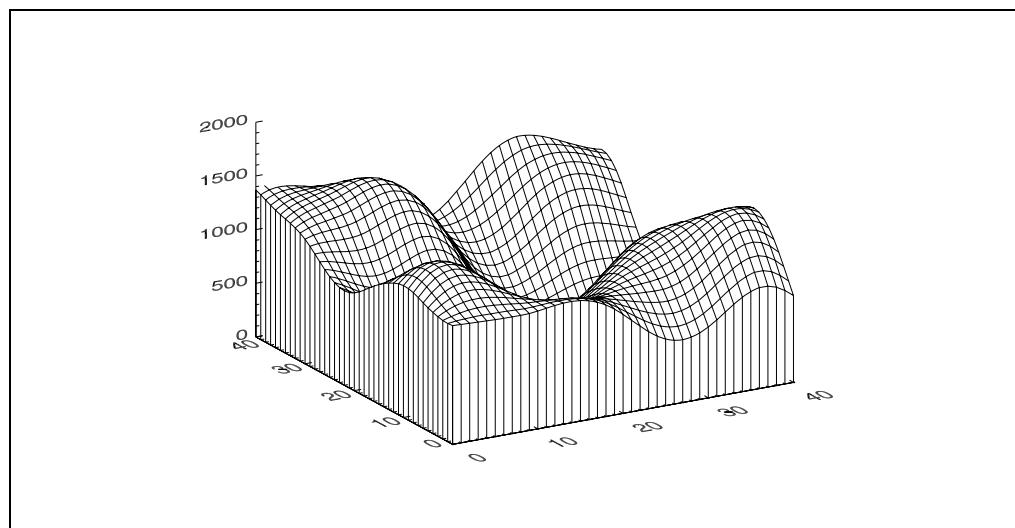


Figure 14: The surface plot with a skirt on the surface.

You can obtain a kind of “stacked line plot” appearance by drawing only horizontal lines. For example, type:

```
IDL> Surface, peak, /Horizontal
```

If you like, you can display just the lower or just the upper surface and not both (which is the default) using keywords. For example, type the following two commands:

```
IDL> Surface, peak, /Upper_Only  
IDL> Surface, peak, /Lower_Only
```

Sometimes you might want to draw just the surface itself with no axes:

```
IDL> Surface, peak, XStyle=4, YStyle=4, ZStyle=4
```

Creating Shaded Surface Plots

It is equally easy to create shaded surface plots of your data. To create a shaded surface plot using a Gouraud light source shading algorithm, type:

```
IDL> Shade_Surf, peak
```

The *Shade_Surf* command accepts most of the keywords accepted by the *Surface* command. So, for example, if you want to rotate the shaded surface, you can type this:

```
IDL> Shade_Surf, peak, lon, lat, Az=45, Ax=30
```

Your output should look similar to the illustration in Figure 15.

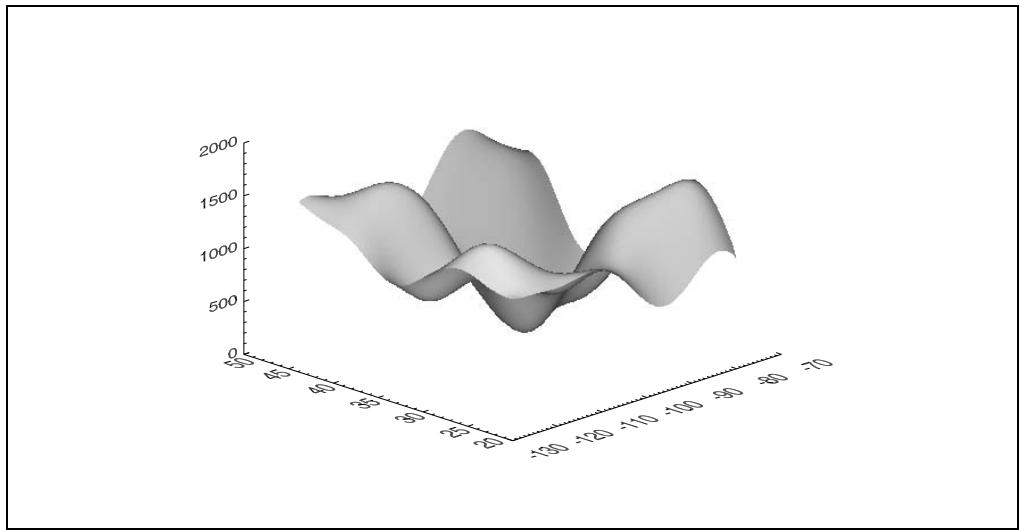


Figure 15: A shaded surface plot of the elevation data using a Gouraud light source shading algorithm.

Changing the Shading Parameters

The shading parameters used by the *Shade_Surf* command can be changed with the *Set_Shading* command. For example, to change the light source from the default of $[0,0,1]$, in which the light rays are parallel to the Z axis, to $[1,0,0]$, in which the light rays are parallel to the X axis, type:

```
IDL> Set_Shading, Light=[1,0,0]
IDL> Shade_Surf, peak
```

You can also indicate which color table indices should be used from the color table for shading purposes. For example, suppose you wanted to load the Red Temperature color table (color table 3) into color indices 100 to 199 and use these for the shading colors. You could type this:

```
IDL> LoadCT, 3, NColors=100, Bottom=100
IDL> Set_Shading, Values=[100, 199]
IDL> Shade_Surf, peak
```

Be sure to set the light source location and color value parameters back to their original values, or you will be confused as you move on with the exercises.

```
IDL> LoadCT, 5
IDL> Set_Shading, Light=[0,0,1], Values=[0,!D.Table_Size-1]
```

Using Another Data Set For the Shading Values

Like with the *Surface* command, another data set can provide the shading values with which to color the first. As before, you use the *Shades* keyword to specify the color index of each point on the surface. The shading of each pixel is interpolated from the surrounding data set values. For example, here is what the *snow* variable looks like as a shaded surface. Type:

```
IDL> Shade_Surf, snow
```

Now use this data set to shade the original elevation data. Type:

```
IDL> Shade_Surf, peak, Shades=BytScl(snow, Top=!D.Table_Size)
```

Your output should look like the illustration in Figure 16.

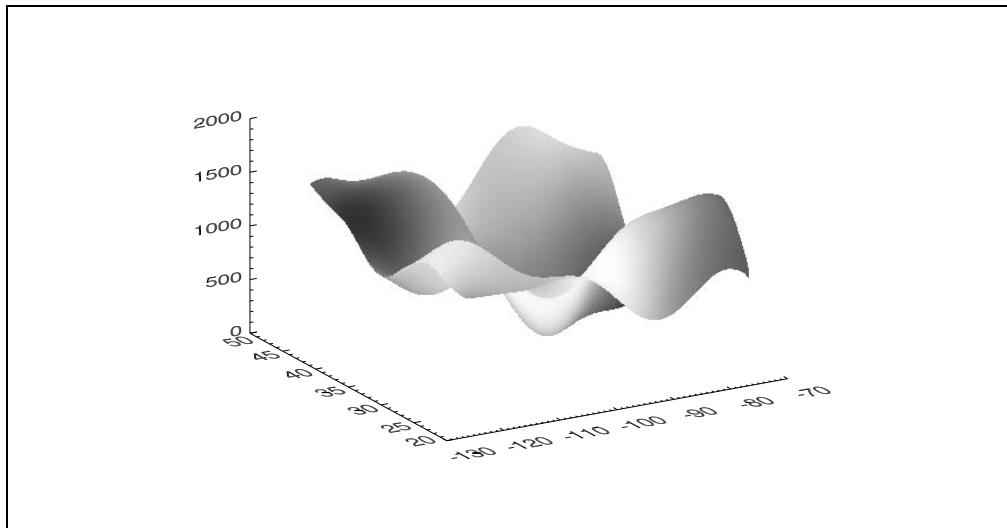


Figure 16: The peak elevation data shaded with the snow data set.

If you want to shade the surface according to its elevation values, simply byte scale the data set itself, type this:

```
IDL> Shade_Surf, peak, Shades=BytScl(peak, Top=!D.Table_Size)
```

Draping another data set over a surface is a way to add an extra dimension to your data. For example, by draping a data set on a three-dimensional surface, you are visually representing four-dimensional information. If you were to animate these two data sets over time you would be visually representing five-dimensional information. (To learn about data animation see “Animating Data in IDL” on page 102.)

Sometimes you just want to drape the original surface on top of the shaded surface. This is easy to do simply by combining the *Shade_Surf* and *Surface* commands. For example, type:

```
IDL> Shade_Surf, peak  
IDL> Surface, peak, /NoErase
```

Creating Contour Plots

Any two-dimensional data set can be displayed in IDL as a contour plot with a single *Contour* command. You can use the *peak* variable if you already have it defined in this IDL session. If not, use the *LoadData* command to load the *Elevation Data* data set, type this:

```
IDL> peak = LoadData(2)  
IDL> Help, peak
```

This data can be viewed as a contour plot with a single command:

```
IDL> Contour, peak, CharSize=1.5
```

Notice that if the *Contour* command is given just the single 2D array as an argument it plots the data set as a function of the number of elements (41 in each direction) in the 2D array. But, as you did before with the *Surface* command, you can specify values for the X and Y axes that may have physical meaning for you. For example, you can use the longitudinal and latitudinal vectors from before, like this:

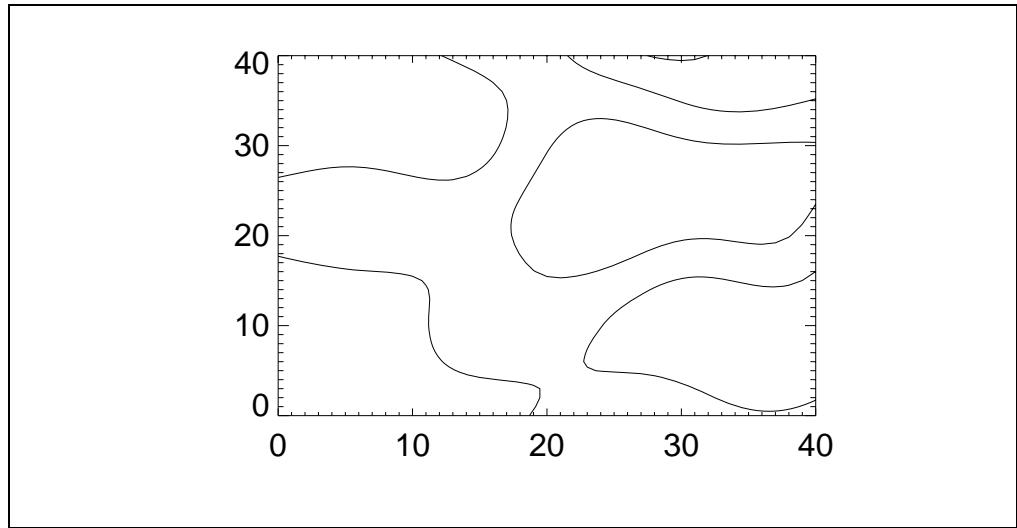


Figure 17: A basic contour plot of the data. Notice that the X and Y axis labels represent the number of elements in the data array.

```
IDL> lat = FIndGen(41) * (24./40) + 24
IDL> lon = FindGen(41) * 50.0/40 - 122
IDL> Contour, peak, lon, lat, XTitle='Longitude', $
      YTitle='Latitude'
```

Notice that the axes are being autoscaled. You can tell this in a couple of ways. First, the contour lines do not go to the edges of the contour plot, and you can see that the labels on the axes are not the same as the minimum and maximum values of the *lon* and *lat* vectors.

```
IDL> Print, Min(lon), Max(lon)
IDL> Print, Min(lat), Max(lat)
```

To prevent autoscaling of the axes, set the *XStyle* and *YStyle* keywords, like this:

```
IDL> Contour, peak, lon, lat, XTitle='Longitude', $
      YTitle='Latitude', XStyle=1, YStyle=1
```

You see the result of this command in the illustration in Figure 18.

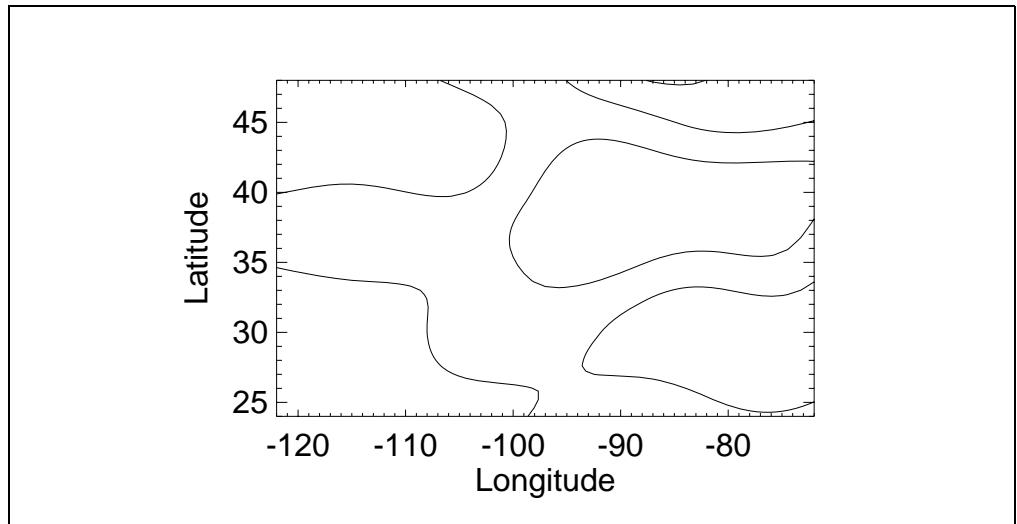


Figure 18: A contour plot with axes representing physically meaningful quantities.

In earlier versions of IDL, the *Contour* command used what was called the *cell drawing method* of calculating and drawing the contours of the data. In this method the contour plot was drawn from the bottom of the contour plot to the top. This method was efficient, but it didn't allow options like contour labeling. The *cell following method* was used to draw each contour line entirely around the contour plot. This took more time, but allowed more control over the contour line. For example, it could be interrupted to allow the placement of a contour label. The cell following method could be selected with the *Follow* keyword:

```
IDL> Contour, peak, lon, lat, XStyle=1, YStyle=1, /Follow
```

Starting in IDL 5 the contour command always uses the cell following method to draw contours, so the *Follow* keyword is obsolete. But it is still used for its beneficial side-effect of labeling every other contour line automatically.

Selecting Contour Levels

By default, IDL selects six regularly spaced contour intervals (five contour lines) to contour, but you can change this in several ways. For example, you can tell IDL how many contour levels to draw with the *NLevels* keyword. IDL will calculate that number of regularly spaced contour intervals. For example, to contour a data set with 12 regularly spaced contours, type:

```
IDL> Contour, peak, lon, lat, XStyle=1, YStyle=1, /Follow, $  
      NLevels=12
```

Your output should look similar to the illustration in Figure 19. You can choose up to 29 contour levels.

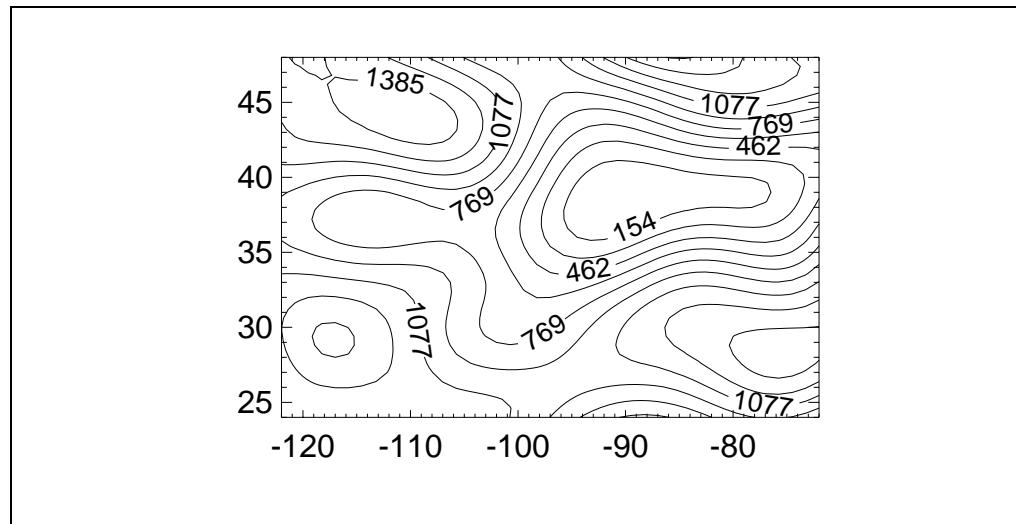


Figure 19: A contour plot with the number of contour levels set to 12. Note that every other contour level is labeled. This is a side-effect of using the *Follow* keyword.

Unfortunately, although the IDL documentation claims that IDL will select a specified number of “equally spaced” contour intervals, this may not happen. If you look closely at the contour plot you just made, you will notice that fewer than 12 levels are calculated by IDL. Apparently, the *NLevels* keyword value is used as a “suggestion” by the contour-selecting algorithm in IDL.

Thus, most IDL programmers choose to calculate the contour levels themselves. For example, you can specify exactly where the contours should be drawn and pass them

to the *Contour* command with the *Levels* keyword, rather than with the *NLevels* keyword, like this:

```
IDL> vals = [200, 300, 600, 750, 800, 900, 1200, 1500]
IDL> Contour, peak, lon, lat, XStyle=1, YStyle=1, /Follow, $
      Levels=vals
```

To choose 12 equally spaced contour intervals, you can write code something like this:

```
IDL> nlevels = 12
IDL> step = (Max(peak) - Min(peak)) / nlevels
IDL> vals = Indgen(nlevels) * step + Min(peak)
IDL> Contour, peak, lon, lat, XStyle=1, YStyle=1, /Follow, $
      Levels=vals
```

If you like, you can specify exactly which contour line should be labeled with the *C_Labels* keyword. This keyword is a vector with as many elements as there are contour lines. (If the number of elements does not match the number of contour lines, the elements do not get re-cycled like they sometimes do with other contour keywords.) If the value of the element is 1 (or more precisely, if it is positive), the contour label is drawn; if the value of the element is 0, the contour label is not drawn. If there is no value for a particular contour line, the line is not labeled. For example, to label the first, third, sixth and seventh contour line, type:

```
IDL> Contour, peak, lon, lat, XStyle=1, YStyle=1, /Follow, $
      Levels=vals, C_Labels=[1,0,1,0,0,1,1,0]
```

To label all the contour lines, you could use the *Replicate* command to replicate the value 1 as many times as you need. For example, type this:

```
IDL> Contour, peak, lon, lat, XStyle=1, YStyle=1, /Follow, $
      Levels=vals, C_Labels=Replicate(1, nlevels)
```

Modifying a Contour Plot

A contour plot can be modified with the same keywords you used for the *Plot* and *Surface* commands. But there are also a number of keywords that apply only to the *Contour* command. These most often modify the contour lines themselves.

For example, to annotate a contour plot with axis titles, you can type:

```
IDL> Contour, peak, lon, lat, XStyle=1, YStyle=1, /Follow,
      XTitle='Longitude', YTitle='Latitude', $
      CharSize=1.5, Title='Study Area 13F89', NLevels=10
```

But, you can also specify annotations on the contour lines themselves by using the *C_annotation* keyword. For example, you could label each contour line with a string label by typing:

```
IDL> Contour, peak, lon, lat, XStyle=1, YStyle=1, /Follow, $
      XTitle='Longitude', YTitle='Latitude', $
      CharSize=1.5, Title='Study Area 13F89', $
      C_annotation=['Low','Middle','High'], $
      Levels=[200, 500, 800]
```

Your output should look similar to the illustration in Figure 20.

Changing the Appearance of a Contour Plot

There are any number of ways to modify the appearance of a contour plot. Here are a few of them. One characteristic you can change is the style of the contour lines. (See

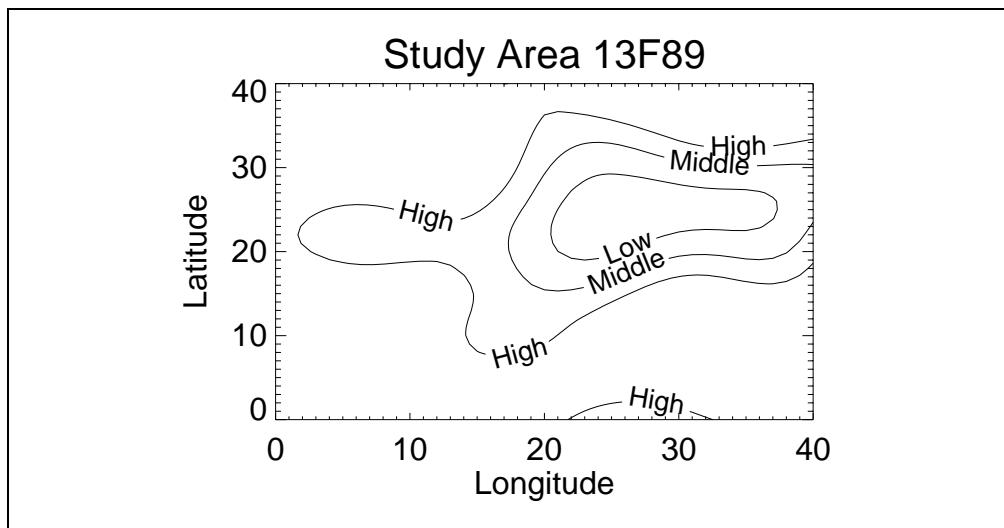


Figure 20: Contour lines can be labeled with text you provide yourself.

Table 3 for a list of possible line style values.) For example, to make the contour lines dashed, type:

```
IDL> Contour, peak, lon, lat, XStyle=1, YStyle=1, $  
      /Follow, CLineStyle=2
```

If you wanted every third line to be dashed, you could specify a vector of line style indices with the *C_LineStyle* keyword. If there are more contour lines than indices, the indices will be recycled or used over. Type:

```
IDL> Contour, peak, lon, lat, XStyle=1, YStyle=1, /Follow, $  
      NLevels=9, CLineStyle=[0,0,2]
```

Your output should look similar to the illustration in Figure 21.

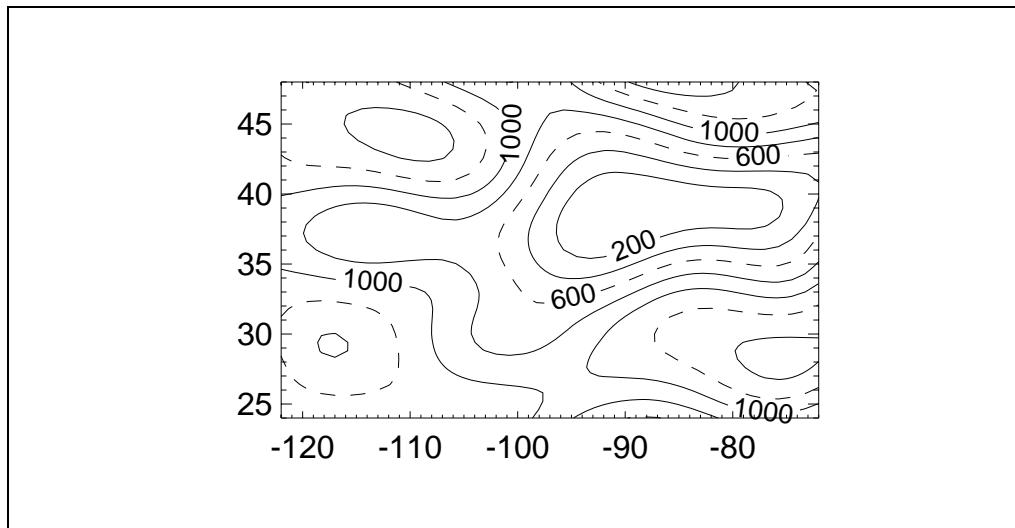


Figure 21: You can modify many aspects of a contour plot. Here every third contour line is drawn in a dashed line style.

The thickness of the contour lines can also be changed. For example, to make all contour lines double thick, type:

```
IDL> Contour, peak, lon, lat, XStyle=1, YStyle=1, $
```

```
NLevels=10, C_Thick=2, /Follow
```

You could make every other line thick by specifying a vector of line thicknesses, like this:

```
IDL> Contour, peak, lon, lat, XStyle=1, YStyle=1, $  
NLevels=12, C_Thick=[1,2], /Follow
```

You can modify the contour plot so that you can easily see the downhill direction. For example, type:

```
IDL> Contour, peak, lon, lat, XStyle=1, YStyle=1, $  
/Follow, NLevels=12, /Downhill
```

Your output should be similar to the illustration in Figure 22.

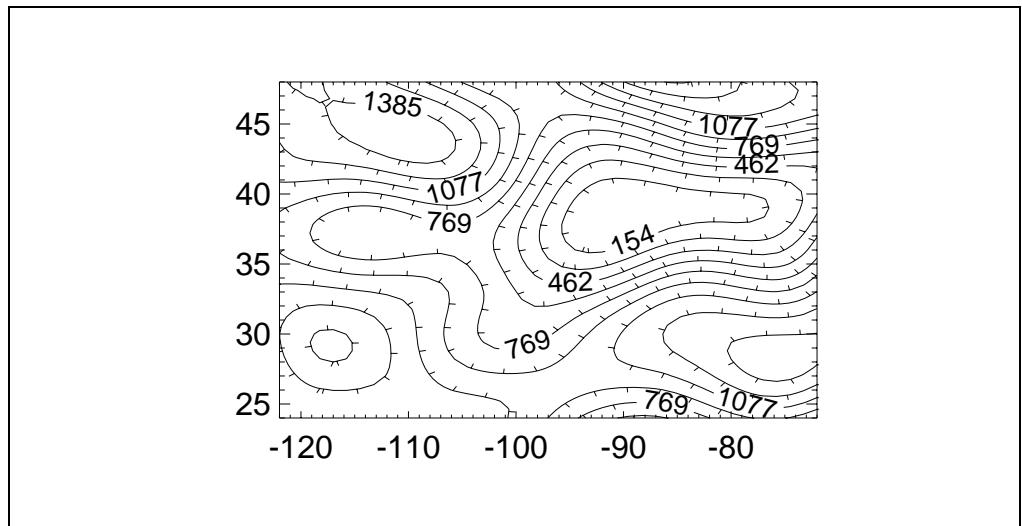


Figure 22: The *Downhill* keyword is set to show the downhill direction of the contour plot.

Adding Color to a Contour Plot

There are many ways to add color to a contour plot. (Colors are discussed in detail in “Working with Colors in IDL” on page 81. For now, just type the *TvLCT* command below. You will learn exactly what the command means later. Basically, you are loading three color vectors that describe the red, green, and blue components of the color triples that describe the charcoal, yellow, and green colors.) For example, if you want a yellow contour plot on a charcoal background, you can type:

```
IDL> TvLCT, [70, 255], [70, 255], [70, 0], 1  
IDL> Device, Decomposed=0  
IDL> Contour, peak, lon, lat, XStyle=1, YStyle=1, $  
NLevels=10, Color=2, Background=1, /Follow
```

You can also color the individual contour lines using the *C_Colors* keyword. For example, if you wanted the contour lines in the plot above to be drawn in green, you can type:

```
IDL> TvLCT, 0, 255, 0, 3  
IDL> Contour, peak, lon, lat, XStyle=1, YStyle=1, /Follow, $  
NLevels=10, Color=2, Background=1, C_Colors=3
```

The *C_Colors* keyword can also be expressed as a vector of color table indices, which will be used in a cyclical fashion to draw the contour lines. For example, you can use

the *Tek_Color* command to create and load drawing colors for the contour lines, like this:

```
IDL> Tek_Color
IDL> TvLCT, [70, 255], [70, 255], [70, 0], 1
IDL> Contour, peak, lon, lat, XStyle=1, YStyle=1, $
      NLevels=10, Color=2, Background=1, $
      C_Colors=IndGen(10)+2, /Follow
```

It is also easy to use the *C_Colors* keyword to make every third contour line blue, while the rest are green. For example, type:

```
IDL> Contour, peak, lon, lat, XStyle=1, YStyle=1, $
      NLevels=12, Color=2, Background=1, $
      C_Colors=[3, 3, 4], /Follow
```

Creating a Filled Contour Plot

Sometimes you don't just want to see contour lines, but you would like to see a filled contour plot. To create a filled contour plot, just use the *Fill* keyword. Here you first load 12 colors into the color table for the fill colors. The color indices will be specified with the *C_Colors* keyword. For example, type:

```
IDL> LoadCT, 0
IDL> LoadCT, 4, NColors=12, Bottom=1
IDL> Contour, peak, lon, lat, XStyle=1, YStyle=1, /Fill, $
      NLevels=12, /Follow, C_Colors=Indgen(12)+1
```

Although it is not obvious from the display, there are problems with doing filled contours in this manner. In fact, there is a "hole" in this contour plot that is filled with the background color. You can see it more easily if you reverse the background and plotting colors. (This is what is done in PostScript, as a matter of fact, and what causes many IDL programmers to pull their hair in frustration.)

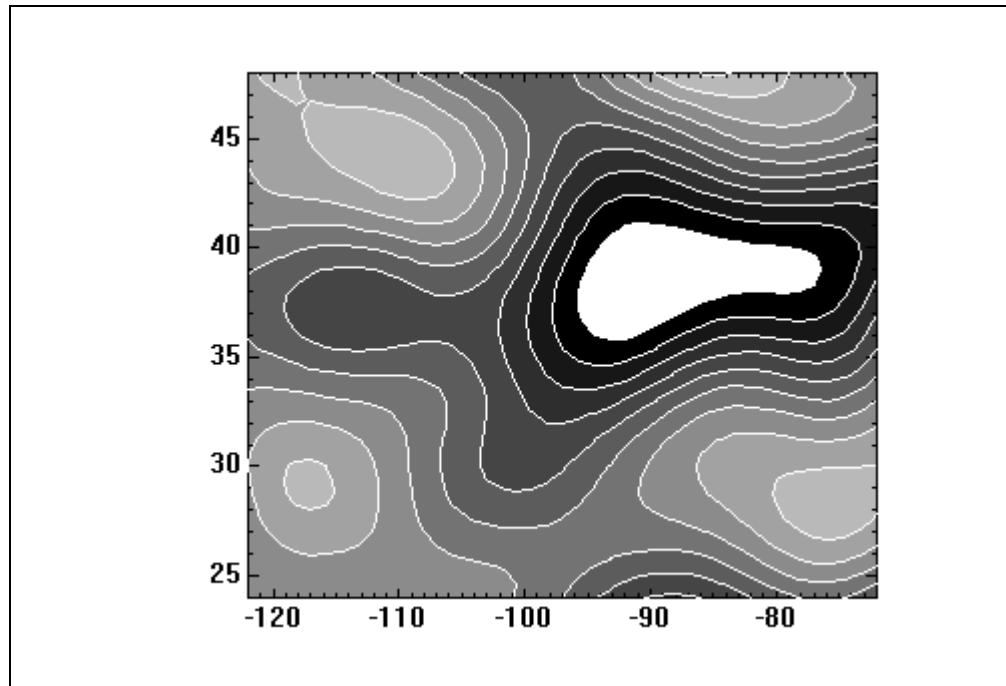


Figure 23: The contour plot showing the "hole" at the lowest contour level.

```
IDL> Contour, peak, lon, lat, XStyle=1, YStyle=1, /Fill, $
```

```
NLevels=12, /Follow, C_Colors=Indgen(12)+1, $  
Background=!P.Color, Color=!P.Background
```

The reason for the hole is that IDL fills the space between the first and second contour lines with the first fill color. It would seem to make more sense to fill the space between the 0th (or background) and first contour with the first fill color. But to get IDL to do that you have to specify your own contour intervals and pass them to the *Contour* command with the *Levels* keyword. This is usually done with code like this:

```
IDL> step = (Max(peak) - Min(peak)) / 12.0  
IDL> clevels = IndGen(12)*step + Min(peak)
```

Now you get the contour fill colors correct:

```
IDL> Contour, peak, lon, lat, XStyle=1, YStyle=1, /Fill, $  
      Levels=clevels, /Follow, C_Colors=Indgen(12)+1, $  
      Background=!P.Color, Color=!P.Background
```



In general, it is a good idea to *always* define your own contour levels when you are working with filled contour plots. Moreover, if you are displaying your filled contour plots along with a color bar, creating your own contour levels is the only way to make sure that your contour levels and the color bar levels are identical.

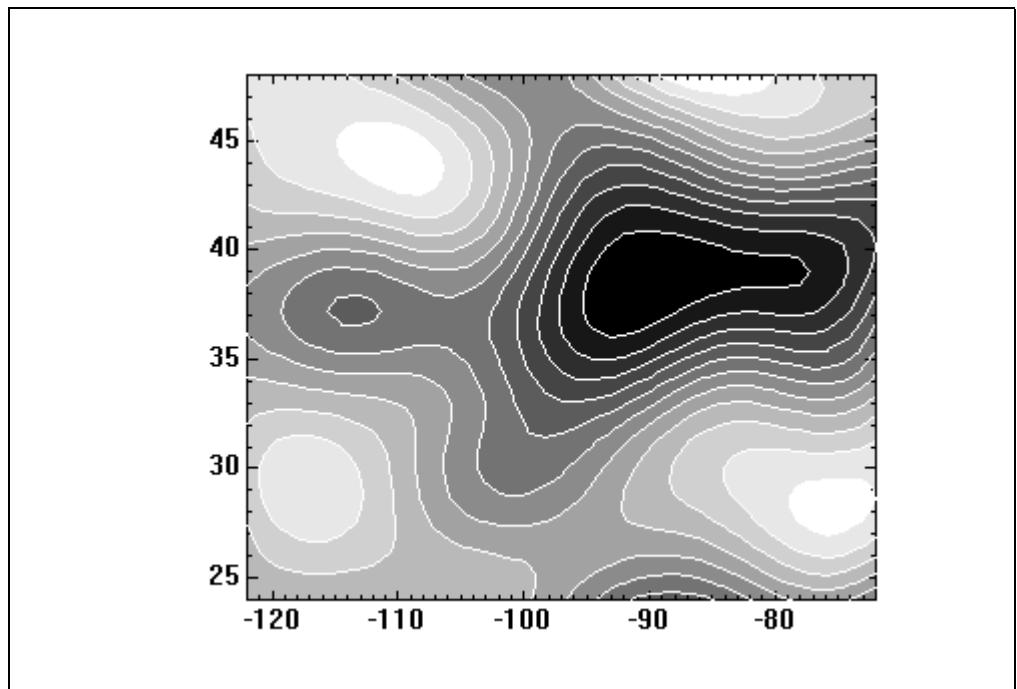


Figure 24: The same contour plot, but with the levels calculated and specified directly. The hole is now filled and the correct number of levels is apparent in the plot.

Sometimes you will want to fill a contour plot that has missing data or contours that extend off the edge of the contour plot. These are called *open contours*. IDL can sometimes have difficulty with open contours. The best way to fill contours of this type, is to use the *Cell_Fill* keyword rather than the *Fill* keyword. This causes the *Contour* command to use a cell filling algorithm, which is not as efficient as the algorithm used by the *Fill* keyword, but which gives better results under these circumstances.

```
IDL> Contour, peak, lon, lat, XStyle=1, YStyle=1, $  
      Levels=clevels, C_Colors=Indgen(12)+1, /Cell_Fill
```



The *Cell_Fill* keyword should also be used if you are putting your filled contour plots on map projections. Otherwise, the contour colors will sometimes be incorrect.

The cell filling algorithm sometimes damages the contour plot axes. You can repair them by drawing the contour plot over again without the data. Like this:

```
IDL> Contour, peak, lon, lat, XStyle=1, YStyle=1, $  
      Levels=clevels, /NoData, /NoErase, /Follow
```

Sometimes you want to see the contour lines on top of the color-filled contours. This is easily accomplished in IDL with the *Overplot* keyword to the *Contour* command. For example, type:

```
IDL> Contour, peak, lon, lat, XStyle=1, YStyle=1, $  
      Levels=clevels, /Fill, C_Colors=IndGen(12)+1  
IDL> Contour, peak, lon, lat, XStyle=1, YStyle=1, /Follow, $  
      Levels=clevels, /Overplot
```

Your output should look similar to the illustration in Figure 25.

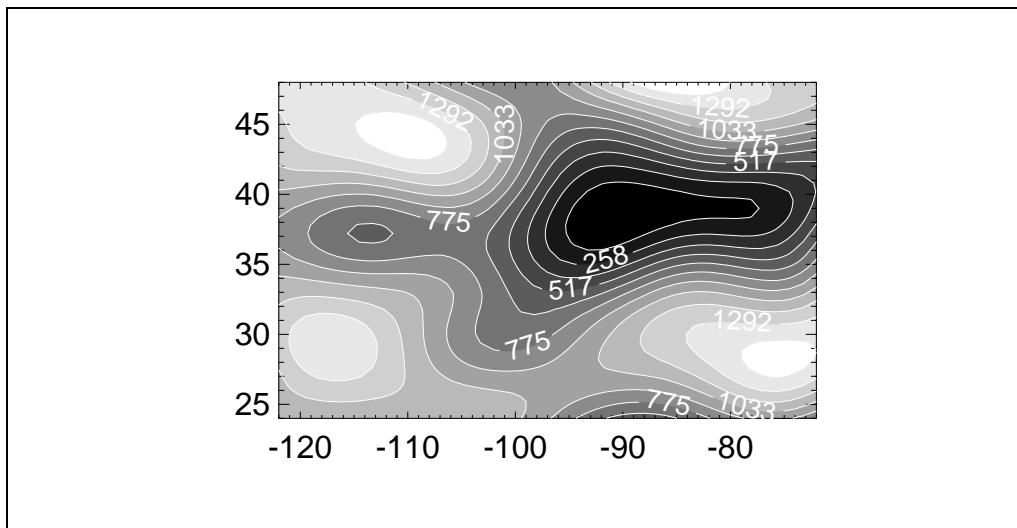


Figure 25: A contour plot with the contour lines overplotted on top of the filled contours.



Don't confuse the *Overplot* keyword with the *NoErase* keyword. They are similar, but definitely not the same. In a contour plot, the *Overplot* keyword draws the contour lines only, *not* the contour axes. The *NoErase* keyword draws the entire contour plot without first erasing what is already on the display.

Positioning Graphic Output in the Display Window

IDL has several ways to position line plots, surface plots, contour plots and other graphical plots (map projections, for example) in the display window. To understand how IDL positions graphics, however, it is important to understand a few definitions. The *graphic position* is that location in the display window that is enclosed by the axes of the plot. The graphic position does not include axes labels, titles, or other annotation (see Figure 26 below). The *graphic region* is that portion of the display window that includes the graphic position, but it also includes space around the graphic position for such things as axes labels, plot and axes titles, etc. The *graphic margin* is defined as the area in the display window that is *not* the graphic position.

The graphic position may be set by *!P.Position* system variable or by the *Position* keyword to the *Plot*, *Surface*, *Contour*, or any of the other IDL graphics commands.

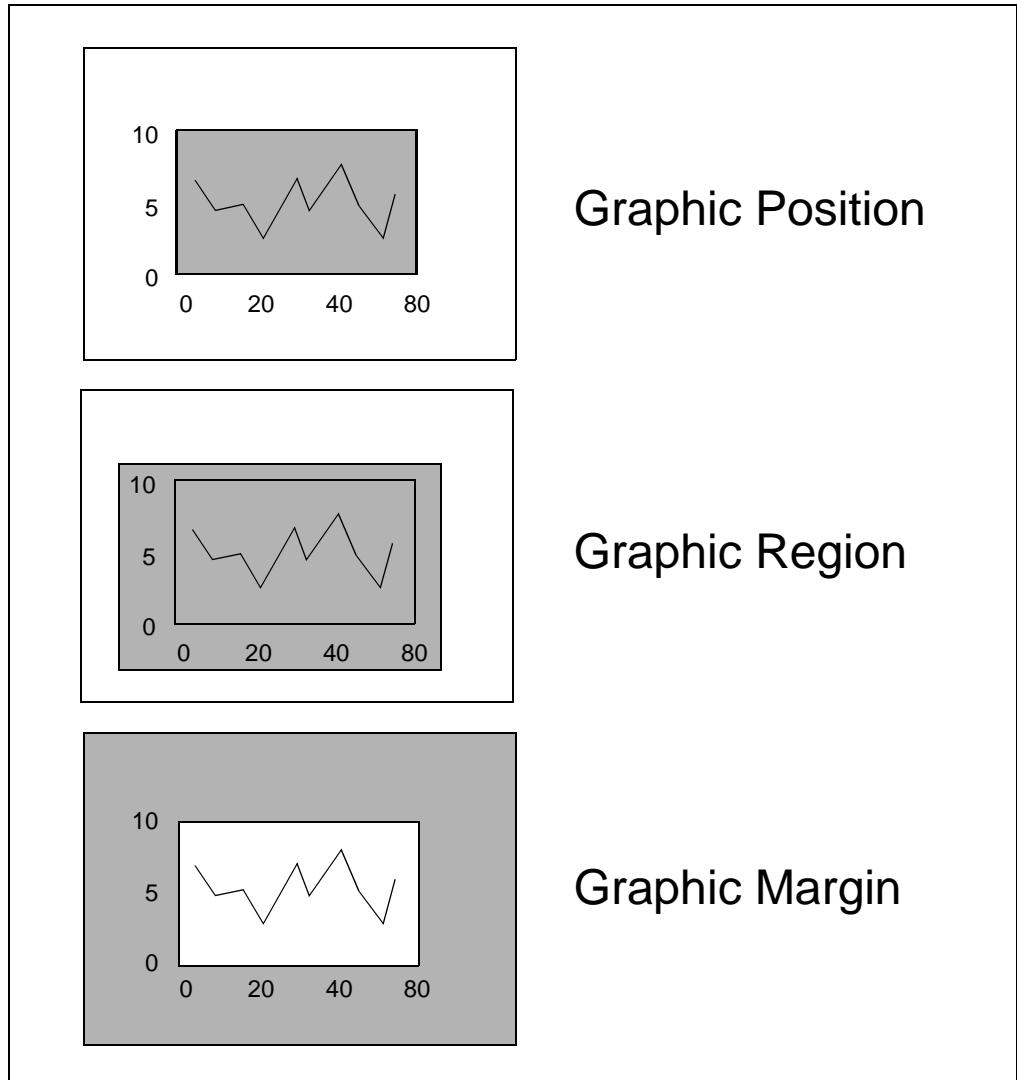


Figure 26: The graphic position is the area of the display enclosed by the axes of the graphic. The graphic region is similar, but also contains space for the graphic's titles and other annotation. The graphic margin is just the opposite of the graphic position. The graphic margin is specified in character units, whereas the graphic position and graphic region are specified in normalized coordinate units.

The entire graphic region may be set by the `!P.Region` system variable, or regions on individual axes can be set using the `Region` field to the `!X`, `!Y`, and `!Z` system variables. The graphic margins may be set by the `[XYZ]Margin` keywords to the `Plot`, `Surface`, `Contour`, or other IDL graphics command or by the `Margin` field in the `!X`, `!Y`, and `!Z` system variables.

By default, IDL sets the graphic margin when it tries to put a graphic into the display window. But, as you will see, that is not always the best choice. It is sometimes better to use the graphic position to position your graphics displays, especially if you are combining graphics output commands in one display window.

Setting the Graphic Margins

The graphic margins can be set with the *[XYZ]Margin* keywords in the graphics command or by setting the *Margin* field of the *!X*, *!Y*, and *!Z* system variables. What is unusual about the graphic margins is that the units are specified in terms of *character size*. The X margin is set by a two-element vector specifying the left and right offsets, respectively, of the axes from the edge of the display. The Y margin specifies the bottom and top offsets in a similar way. The default margins are 10 and 3 for the X axis and 4 and 2 for the Y axis. To see what the current character sizes are in device or pixel units, type:

```
IDL> Print, !D.X_Ch_Size, !D.Y_Ch_Size
```

On a Macintosh, for example, the default character size is 6 pixels in X and 9 pixels in Y. Thus, a contour plot is drawn with (6 times 10) 60 pixels on its left edge of the plot and (6 times 3) 18 pixels on its right edge. If the *CharSize* keyword is set to 2 on a *Contour* command, for example, then there will be 120 pixels on the left edge of the plot and 36 on the right edge.

For example, to change the graphic margins to three default character units all the way around the plot, you would type:

```
IDL> Plot, time, curve, XMargin=[3,3], YMargin=[3,3]
```

Notice, however, that the plot looks very different if you also change the character size, since graphic margins are specified in terms of character size. Try this:

```
IDL> !X.Margin = [3,3]
IDL> !Y.Margin = [3,3]
IDL> Contour, peak, CharSize=2.5
IDL> Contour, peak, CharSize=1.5
```

If you play around a bit with other character sizes, you can see that at larger character sizes, the characters get very large and the graphics portion of the plot gets very small. This is not always what you want.

Be sure to set the graphic margins back to their default values before you move on. Type:

```
IDL> !X.Margin = [10,3]
IDL> !Y.Margin = [4,2]
```



Unlike many other system variables, whose values are set back to the default values by setting the system variable equal to 0, the margin system variables must be set explicitly to their default values. If you didn't type the two commands above, be sure to do it now.

Setting the Graphic Position

Setting the graphic position requires setting a four-element vector giving, in order, the coordinates *[X0, Y0, X1, Y1]* of the lower-left and upper-right corners of the graphic in the display window. These coordinates are always expressed in normalized units ranging from 0 to 1 (i.e., 0 is always either the left or the bottom of the display window, and 1 is always either the right or top of the display window).

For example, suppose you wanted your graphic output to be displayed in the top half of the display window. Then you might set the *!P.Position* system variable and display your graphic like this:

```
IDL> !P.Position = [0.1, 0.5, 0.9, 0.9]
IDL> Plot, time, curve
```

All subsequent graphics output will be positioned similarly. To reset the *!P.Position* system variable so that subsequent graphic output fills the window normally, type:

```
IDL> !P.Position = 0
```

If you wanted to position just one graphic display, you could specify a graphic position with the *Position* keyword to the graphics command. Suppose you wanted a contour plot to just fill up the left-hand side of a display window. You might type this:

```
IDL> Contour, peak, Position=[0.1, 0.1, 0.5, 0.9]
```



Note that the *Position* keyword can be used to put multiple graphics plots into the same display window. Just be sure to use the *NoErase* keyword on the second and all subsequent graphics commands. This will prevent the display from being erased first, which is the default behavior for all graphics output commands except *TV* and *TVScl*. For example, to put a line plot above a contour plot, you can type:

```
IDL> Plot, time, curve, Position=[0.1, 0.55, 0.95, 0.95]
IDL> Contour, peak, Position=[0.1, 0.1, 0.95, 0.45], /NoErase
```

Setting the Graphic Region

The graphic region is specified in normalized coordinates in the same way as the graphic position, and can be specified by setting the *!P.Region* system variable. Since there is not an equivalent keyword that can be set on a graphics command, setting the graphics region is often less convenient than setting the graphics position. Be sure you reset the system variable if you want subsequent plots to use the whole display window normally. For example, to display a plot in the upper two-thirds of your display window, type:

```
IDL> !P.Region = [0.1, 0.33, 0.9, 0.9]
IDL> Plot, time, curve
```

To reset the *!P.Region* system variable so that subsequent plots fill the window normally, type:

```
IDL> !P.Region = 0
```

Creating Multiple Graphics Plots

As you can see, multiple plots can be positioned on the display using the graphic position and graphic region system variables and keywords discussed above (as long as the second and all subsequent plots use the *NoErase* keyword). But it is much easier to use the *!P.Multi* system variable to create multiple plots on the display. *!P.Multi* is defined as a five element vector as follows:

!P.Multi[0]

The first element of *!P.Multi* contains the number of graphics plots *remaining to plot* on the display or PostScript page. This is a little non-intuitive, but you will see how it is used in a moment. It is normally set to 0, meaning that as there are no more graphics plots remaining to be output on the display, the next graphic command will erase the display and start with the first of the multiple graphics plots.

!P.Multi[1]

This element specifies the number of graphics columns on the page.

!P.Multi[2]

This element specifies the number of graphics rows on the page.

!P.Multi[3]

This element specifies the number of graphics plots stacked in the Z direction. (This applies only if you have established a 3D coordinate system.)

!P.Multi[4]

This element specifies whether the graphics plots are going be displayed by filling up the rows first (*!P.Multi[4]=0*) or by filling up the columns first (*!P.Multi[4]=1*).

For example, suppose you want to set *!P.Multi* to display four graphics plots on the display in two columns and two rows, and you would like the graphics plots to fill up the columns first. You would type:

```
IDL> !P.Multi = [0, 2, 2, 0, 1]
```

Now as you display each of the four graphics plots, each graphic fits into about a quarter of the display window. Type:

```
IDL> Window, XSize=500, YSize=500
IDL> Plot, time, curve, LineStyle=0
IDL> Contour, peak, lon, lat, XStyle=1, YStyle=1, NLevels=10
IDL> Surface, peak, lon, lat
IDL> Shade_Surf, peak, lon, lat
```

Your output should look similar to the output in Figure 27.

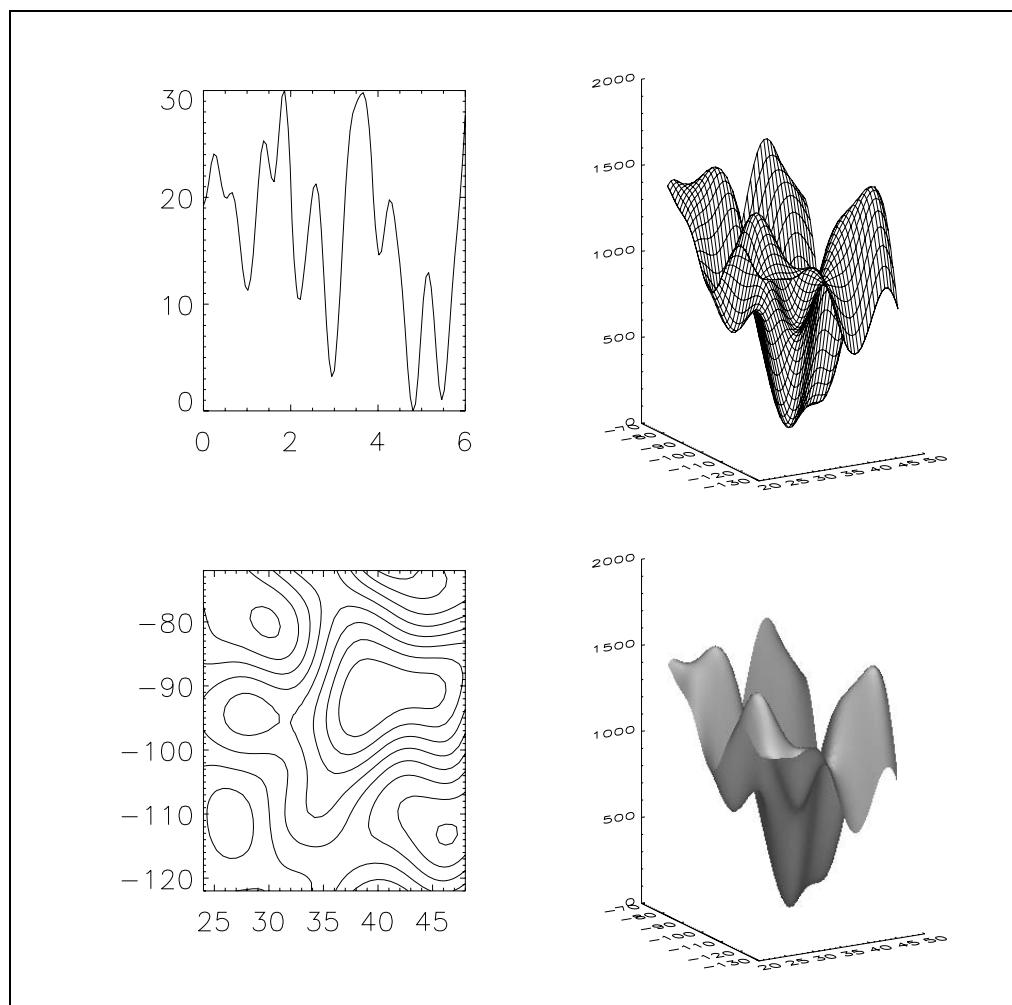


Figure 27: You can plot multiple graphics plots in a single display window.

Leaving Room for Titles with Multiple Graphics Plots

When IDL calculates the position of the graphics plots, it uses the entire window to determine how big each plot should be. But sometimes you would like to have extra

room on the display for titles and other types of annotation. You can leave room with multiple plots by using the “outside margin” fields of the `!X`, `!Y`, and `!Z` system variables. The outside margins only apply when the `!P.Multi` system variable is being used. They are calculated in character units, just as the normal graphic margins are calculated.

For example, suppose you wanted to have an overall title for the four-graphic plot display you just created. You might leave room for the title and create it like this:

```
IDL> !P.Multi = [0, 2, 2, 0, 1]
IDL> !Y.OMargin = [2, 4]
IDL> Plot, time, curve, LineStyle=0
IDL> Contour, peak, lon, lat, XStyle=1, YStyle=1, NLevels=10
IDL> Surface, peak, lon, lat
IDL> Shade_Surf, peak, lon, lat
IDL> XYOutS, 0.5, 0.9, /Normal, 'Four Graphics Plots', $
      Alignment=0.5, CharSize=2.5
```

Your output should look like the illustration in Figure 28.

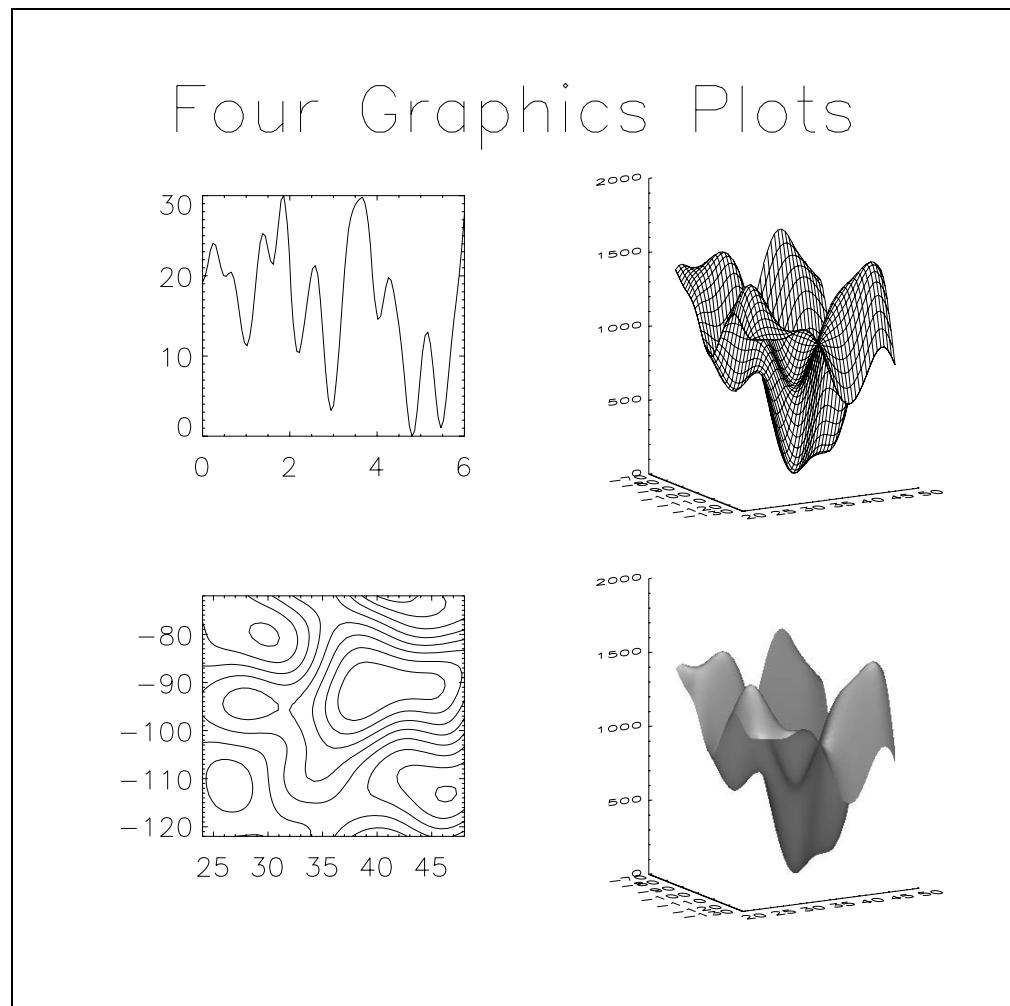


Figure 28: A four by four multiplot with space left at the top of the plot for a title by using the `!Y.OMargin` keyword.

Non-Symmetrical Arrangements with !P.Multi

Plotting with *P.Multi* does not have to be symmetrical on the display. For example, suppose you wanted the surface and shaded surface plot in the left-hand side of the display window one above the other. And you wanted the right-hand side of the page to be filled with a contour plot of the same data. You can type this code:

```
IDL> !P.Multi = [0, 2, 2, 0, 1]
IDL> !Y.OMargin=[0,0]
IDL> Surface, peak, lon, lat
IDL> Shade_Surf, peak, lon, lat
IDL> !P.Multi = [1, 2, 1, 0, 0]
IDL> Contour, peak, lon, lat, XStyle=1, YStyle=1, NLevels=10
```

The first *P.Multi* command sets up a two-column by two-row configuration and the first two of these plots are drawn. The second *P.Multi* command sets up a two-column by one-row configuration. But notice that *P.Multi[0]* is set to 1. This results in the contour plot going into the *second* position in the window instead of the first. The result is shown in Figure 29, below.

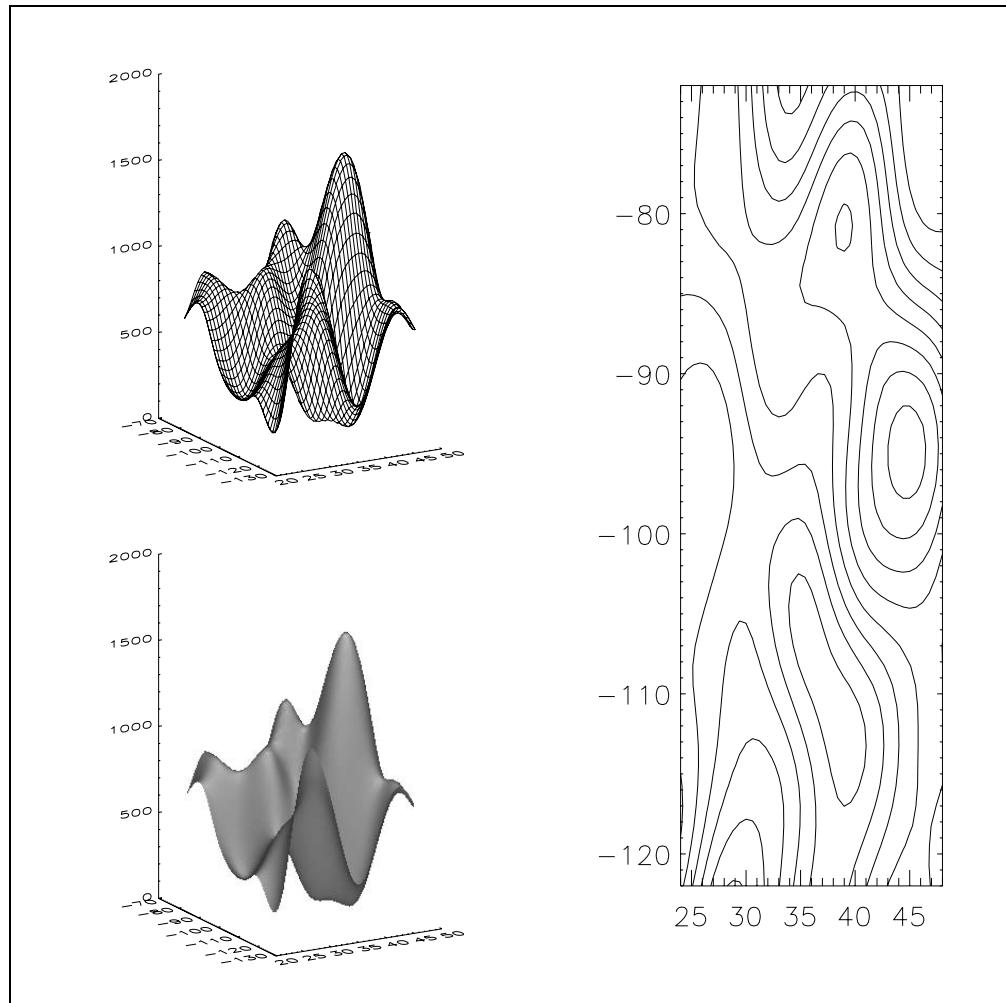


Figure 29: You can use *P.Multi* to position asymmetric arrangements of plots in the display window.



Note that the *TV* command does not work with *P.Multi* like the *Plot* or *Contour* commands do. However, the *TVImage* program, which is a replacement for the *TV* com-

mand and is among the programs you downloaded to use with this book, does honor the *!P.Multi* system variable. Try these commands:

```
IDL> image = LoadData(7)
IDL> !P.Multi=[0, 2, 2]
IDL> FOR j=0,3 DO TVImage, image
```

Be sure you reset *!P.Multi* to display a single graphics plot on the page. Like many system variables, *!P.Multi* can be reset to its default values by setting it equal to zero, like this:

```
IDL> !P.Multi = 0
```

Adding Text to Graphical Displays

Plot annotations and other text can be added to graphical displays in a variety of ways, the most common of which is via keywords to graphical display commands. The text that is added can come in any of three font “flavors” or styles, if you like: *vector fonts* (sometimes called software or Hershey fonts), *true-type fonts*, and *hardware fonts*. The type of font is selected by setting the *!P.Font* system variable or by setting the *Font* keyword on a graphical output command according to Table 6, below.

!P.Font	Font Selection
-1	Vector fonts (also called software or Hershey fonts)
0	Hardware fonts
1	True-type outline fonts

Table 6: *The font “flavor” can be selected by setting the *!P.Font* system variable or the *Font* keyword to the appropriate value. Vector fonts are the default font type for direct graphics commands. They have the advantage of being platform independent.*

By default fonts in direct graphics routines are set to the vector or software font style. Vector fonts are described by vector coordinates. As a result, they are platform independent and can be rotated in 3D space easily. However, many people find vector fonts too “thin” for quality hardcopy output and prefer more substantial fonts for this purpose (i.e., true-type fonts or PostScript hardware printer fonts). Vector fonts can be selected permanently by setting the *!P.Font* system variable to -1 or by setting the *Font* keyword on a graphical output command, like this:

```
IDL> Plot, time, curve, Font=-1, XTitle='Time', $
      YTitle='Signal', Title='Experiment 35F3a'
```

True-type fonts are also called outline fonts. The font is described by a set of outlines that are then filled by creating a set of polygons. IDL comes with four true-type font families: Times, Helvetica, Courier, and Symbol, but you can supplement these with any other true-type font family you have available. True-type fonts take longer to render because the font must be scaled and the polygons created and filled, and many people find them a bit unattractive at small point sizes on normal low-resolution displays. But they have the great advantage of being rotatable and nice looking on hardcopy output devices. True-type fonts are the default font for the object graphics system in IDL.

To render a plot with the default true-type Helvetica font face, set the *Font* keyword to 1, like this:

```
IDL> Plot, time, curve, Font=1, XTitle='Time', $  
      YTitle='Signal', Title='Experiment 35F3a'
```

True-type fonts can be selected with the *Device* command using the *Set_Font* and *TT_Font* keywords, like this:

```
IDL> Device, Set_Font='Courier', /TT_Font  
IDL> Plot, time, curve, Font=1, XTitle='Time', $  
      YTitle='Signal', Title='Experiment 35F3a'
```

To learn more about true-type fonts in IDL, use your on-line help system like this:

```
IDL> ? fonts
```

Hardware fonts are selected by setting the *!P.Font* system variable or the *Font* keyword to 0. Normally hardware fonts are not used on the display, but are used when output is sent to a hardcopy output device such as a PostScript printer. Unfortunately, hardware fonts do not rotate well in 3D space and so were not normally used with 3D commands like the *Surface* command.

```
IDL> Plot, time, curve, Font=0, XTitle='Time', $  
      YTitle='Signal', Title='Experiment 35F3a'
```

Finding the Names of Available Fonts

You can find the names of available hardware fonts by using the *Device* command like this:

```
IDL> Device, Font='*', Get_FontNames=fontnames  
IDL> For j=0,N_Elements(fontnames)-1 DO Print, fontnames[j]
```

The names of true-type fonts are found in a similar way, except that the *TT_Font* keyword is used to select just the available true-type fonts on your system. (You can add your own true-type fonts to the four families supplied with IDL. See the IDL on-line help to find out how.)

```
IDL> Device, Font='*', Get_FontNames=fontnames, /TT_Font  
IDL> For j=0,N_Elements(fontnames)-1 DO Print, fontnames[j]
```

The names of available vector fonts are given in Table 7 on page 53.

Adding Text with the XYOutS Command

A very important command in IDL is the *XYOutS* command (“at an XY location, *OUTput a String*”). It is used to place a text string at a specific location on the display. (The first positional parameter to *XYOutS* is the X location and the second positional parameter is the Y location). For example, to add a larger title to a line plot, you can type commands like this:

```
IDL> Plot, time, curve, Position=[0.15, 0.15, 0.95, 0.85]  
IDL> XYOutS, 0.5, 32, 'Results: Experiment 35F3a', Size=2.0
```

Notice that you specified the X and Y location with *data* coordinates. Also notice that the Y coordinate is outside the boundary of the plot. By default, the *XYOutS* procedure uses the *data* coordinate system, but the *device* and *normalized* coordinate systems can also be used if the appropriate keywords are specified.

(The data coordinate system is, naturally, described by the data itself. Device coordinates are sometimes called pixel coordinates and the device coordinate system is often used with images. A normalized coordinate system goes from 0 to 1 in each direction. Normalized coordinates are often used when you want device-independent graphics output.)

For example, you can put the title on the line plot using *normalized* coordinates, like this. When you are writing IDL programs it is often a good idea to specify things like titles and other annotation with normalized coordinates. This makes it easy to place graphics not only in the display window, but in PostScript and other hardcopy output files as well.

```
IDL> Plot, time, curve, Position=[0.15, 0.15, 0.95, 0.85]
IDL> XYOutS, 0.2, 0.92, 'Results: Experiment 35F3a', $
      Size=2.0, /Normal
```

Number	Description	Number	Description
!3	Simplex Roman	!12	Simplex Script
!4	Simplex Greek	!13	Complex Script
!5	Duplex Roman	!14	Gothic Italian
!6	Complex Roman	!15	Gothic German
!7	Complex Greek	!16	Cyrillic
!8	Complex Italian	!17	Triplex Roman
!9	Math Font	!18	Triplex Italian
!10	Special Characters	!20	Miscellaneous
!11	Gothic English	!X	Revert to entry font

Table 7: *The Hershey fonts with corresponding index number used to select them within IDL.*

Using XYOutS with Vector Fonts

The *XYOutS* command can be used with either vector, true-type, or hardware fonts. Simply set the appropriate *Font* keyword value as described above. The discussion here concerns itself with vector fonts, since that is the font system that is most often used in direct graphics commands. A list of the available vector fonts along with the index numbers you use to specify the specific font is given in Table 7.

The major advantage of vector or Hershey fonts, is that they are platform independent and can be scaled and rotated in 3D space. For example, you can write the plot title above in Triplex Roman characters by typing:

```
IDL> Plot, time, curve, Position=[0.15, 0.15, 0.95, 0.85]
IDL> XYOutS, 0.2, 0.92, '!17Results: Experiment 35F3a!X', $
      Size=2.0, /Normal
```

The Triplex Roman characters were specified with the *!17* escape sequence. The *!X* at the end of the title string caused the font to revert back to the Simplex Roman font that was in effect before you changed it to Triplex Roman. This reversion step is important because otherwise the default font gets set to Triplex Roman and all subsequent strings will be set with Triplex Roman characters. For example, try using a Greek character as the X title of the plot and try writing the plot title like this:

```
IDL> Plot, time, curve, XTitle='!7w', $
      Position=[0.15, 0.15, 0.95, 0.85]
IDL> XYOutS, 0.2, 0.92, 'Experiment 35F3X', Size=2.0, /Normal
```

You can see the results in Figure 30. Notice that now, even though you didn't specify that it should be so, the title is written in Greek characters. The only way to make it

revert back to the default Simplex Roman is write another string with explicit Simplex Roman characters. For example:

```
IDL> XYOutS, 0.5, 0.5, '!3Junk', /Normal, CharSize=-1
```

Notice the use of the *CharSize* keyword in the code above. When this keyword has a value of -1, the text string is suppressed and not written to the display.

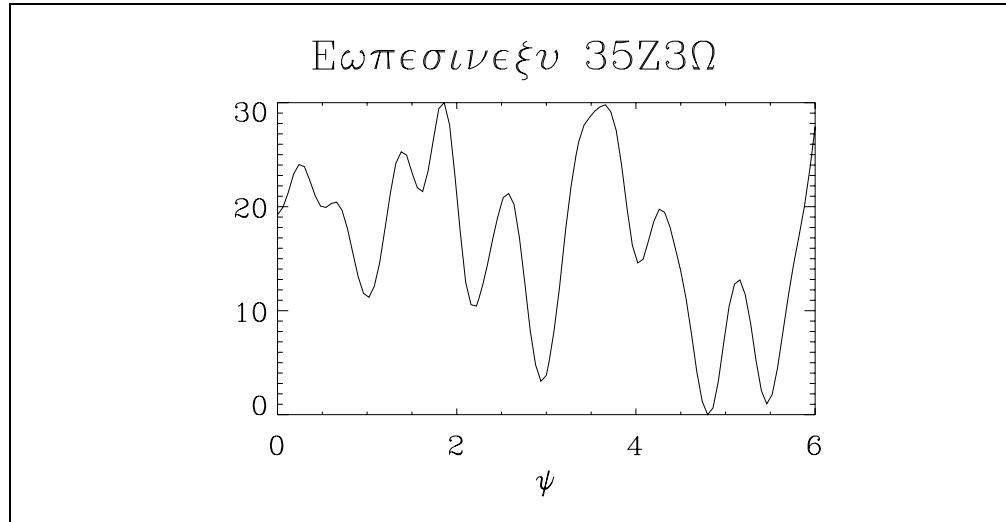


Figure 30: Be careful when you select a Hershey font, or you might end up with plot titles that look like Greek to your users.

Aligning Text

You can position text with respect to the location specified in the call to *XYOutS* with the *Alignment* keyword. A value of 0 will left justify the string (this is the default case); a value of 1 will right justify the string; and a value of 0.5 will center the string with respect to the location specified by the X and Y values. For example:

```
IDL> Window, XSize=300, YSize=250
IDL> XYOutS, 150, 55, 'Research', Alignment=0.0, $
     /Device, CharSize=2.0
IDL> XYOutS, 150, 110, 'Research', Alignment=0.5, $
     /Device, CharSize=2.0
IDL> XYOutS, 150, 170, 'Research', Alignment=1.0, $
     /Device, CharSize=2.0
IDL> PlotS, [0.5, 0.5], [1.0, 0.0], /Normal
```

Erasing Text

Text that is written with *XYOutS* can sometimes be “erased” by writing the same text over in the background color. The *Color* keyword in conjunction with the *!P.Background* system variable is used for this purpose. Note that this only works perfectly if the text was written on nothing but the background! There are often other, better methods to erase annotations. (See “Erasing Annotation From the Display” on page 114, for example.) To see how to erase annotation with the background color, type this:

```
IDL> Window, XSize=300, YSize=250
IDL> XYOutS, 150, 110, 'Research', Alignment=0.50, $
     /Device, CharSize=2.0
IDL> XYOutS, 150, 110, 'Research', Alignment=0.50, $
     /Device, CharSize=2.0, Color=!P.Background
```

Orienting Text

The text specified with the *XYOutS* command can be oriented with respect to the horizontal with the *Orientation* keyword, which specifies the number of degrees the text baseline is rotated away from the horizontal baseline. For example, type:

```
IDL> Window, XSize=300, YSize=250
IDL> XYOutS, 150, 110, 'Research', Alignment=0.50, $
      /Device, CharSize=2.0, Orientation=45
IDL> XYOutS, 150, 180, 'Research', Alignment=0.50, $
      /Device, CharSize=2.0, Orientation=-45
```

Positioning Text

IDL also provides a number of ways to position and manipulate text in graphical displays. For example, it is possible to create subscripts and superscripts in a plot title. Text positioning is accomplished by means of embedded positioning commands, as shown in Table 8, below.

Embedded Command	Command Action
!A	Shift text above the division line.
!B	Shift text below the division line.
!C	Insert a carriable return and begin new line of text.
!D	Create a first-level subscript.
!E	Create a first-level superscript.
!I	Create an index-type subscript
!L	Create a second-level subscript.
!N	Shift back to normal level after changing level.
!R	Restore text position to last saved position.
!S	Save the current text position.
!U	Create an index-type superscript.
!X	Return to entry font. (Used after font changes.)
!Z(u0,...un)	Display one or more characters by their unicode value.
!!	Display the exclamation mark!

Table 8: *The embedded commands that can accomplish text positioning in strings. These are often used to produce subscripts and superscripts in plot titles, for example.*

For example, an equation can be written like this:

```
IDL> XYOutS, 0.5, 0.5, /Normal, Alignment=0.5, Size=3.0, $
      'Y = 3X!U2!N + 5x + 3'
```

Adding Lines and Symbols to Graphical Displays

Another useful routine for annotating graphical displays is the *PlotS* command, which is used to draw symbols or lines on the graphic display. The *PlotS* command works in either two- or three-dimensional space.

To draw a line with the *PlotS* procedure, simply provide it with a vector of X and Y values that describe XY point pairs that should be connected. For example, to draw a baseline on the line plot from the point (0, 15) to the point (6, 15), type:

```
IDL> Window, XSize=500, YSize=400
IDL> Plot, time, curve
IDL> xvalues = [0, 6]
IDL> yvalues = [15, 15]
IDL> PlotS, xvalues, yvalues, LineStyle=2
```

Your output should look similar to the output in Figure 31, below.

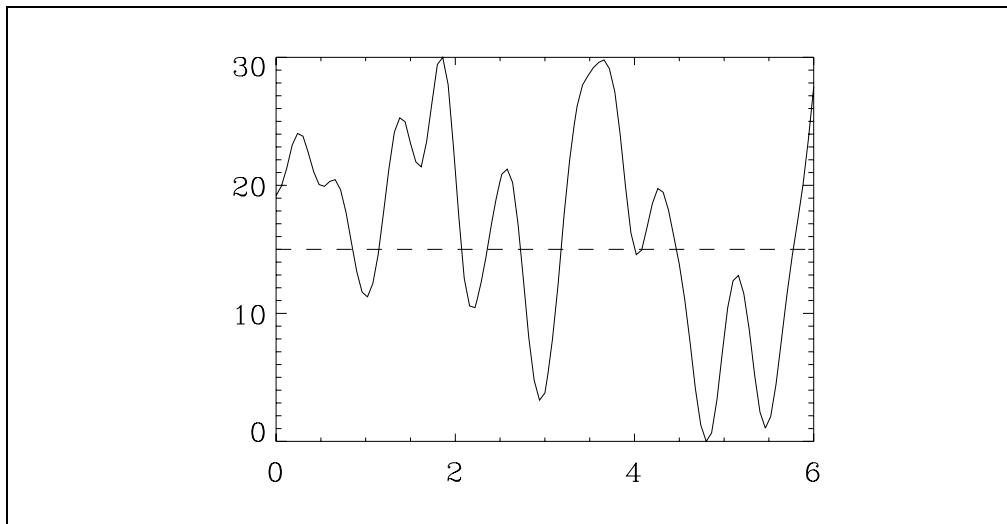


Figure 31: This plot is annotated with a dashed line drawn across its center with the *PlotS* command.

The *PlotS* procedure can also be used to place marker symbols at various locations. For example, here is a way to label every fifth point in the curve with a diamond symbol.

```
IDL> TvLCT, [70, 255, 0], [70, 255, 250], [70, 0, 0], 1
IDL> Plot, time, curve, Background=1, Color=2
IDL> index = IndGen(20)*5
IDL> PlotS, time[index], curve[index], PSym=4, $
      Color=3, SymSize=2
```

The *PlotS* command can also be used to draw a box around important information in a plot. By combining the *PlotS* command with other graphics commands, such as *XYOutS*, you can effectively annotate your graphic displays. For example, like this:

```
IDL> TvLCT, [70, 255, 0], [70, 255, 255], [70, 0, 0], 1
IDL> Device, Decomposed=0
IDL> Plot, time, curve, Background=1, Color=2
IDL> box_x_coords = [0.4, 0.4, 0.6, 0.6, 0.4]
IDL> box_y_coords = [0.4, 0.6, 0.6, 0.4, 0.4]
IDL> PlotS, box_x_coords, box_y_coords, Color=3, /Normal
IDL> XYOutS, 0.5, 0.3, 'Critical Zone', Color=3, Size=2, $
      Alignment = 0.5, /Normal
```



You can easily use the *XYOutS* and *PlotS* commands to create legends for your graphics displays.

Adding Color to Your Graphical Displays

Another useful way to annotate your graphical displays is to use color. The *Polyfill* command is a low-level graphic display command that will fill any arbitrarily shaped polygon (which can be specified in either two or three dimensions) with a particular color or pattern. For example, you can use *Polyfill* to fill the box in the line plot above with a red color:

```
IDL> Tvlct, 255, 0, 0, 4
IDL> Erase, Color=1
IDL> Polyfill, box_x_coords, box_y_coords, Color=4, /Normal
IDL> Plot, time, curve, Background=1, Color=2, /NoErase
IDL> PlotS, box_x_coords, box_y_coords, Color=3, /Normal
IDL> XYOutS, 0.5, 0.3, 'Critical Zone', Color=3, Size=2, $
      Alignment = 0.5, /Normal
```

Color can sometimes serve to represent another dimensional property of a data set. For example, you might display XY data as a 2D plot of circles (polygons), with the color of each polygon representing some additional property of the data such as temperature or population density, etc. Let's see how this can be done.

IDL doesn't have a built-in circle-generator, but it is easy enough to write such a function. Open a text editor and type the code below to create the IDL function named *Circle*.

```
FUNCTION CIRCLE, xcenter, ycenter, radius
points = (2 * !PI / 99.0) * FIndGen(100)
x = xcenter + radius * Cos(points)
y = ycenter + radius * Sin(points)
RETURN, Transpose([[x], [y]])
END
```

The X and Y points that comprise the circle will be returned in a 2 by 100 array. You can use this as input to the *Polyfill* command. Save the program as *circle.pro* and *compile it* by typing this command:

```
IDL> .Compile circle
```

Next, create some randomly-located X and Y data. (You will set the *seed* to begin, so that your results will look like the illustration in Figure 32, below.) Type:

```
IDL> seed = -3L
IDL> x = RandomU(seed, 30)
IDL> y = RandomU(seed, 30)
```

Let the Z values be a function of these X and Y values. Type:

```
IDL> z = (3 * ((x-0.5)^2) + 5 * ((y-0.25)^2)) * 1000
```

Open a window and plot the XY locations, so you can see how the data is distributed in a random pattern. Type:

```
IDL> Window, XSize=400, YSize=350
IDL> Plot, x, y, Psym=4, Position=[0.15, 0.15, 0.75, 0.95], $
      XTitle='X Locations', YTitle='Y Locations'
```

You are going to display the Z data associated with each XY point pair as a circle of a different color. To do this, you will need to load a color table and scale the Z data into the range of colors you have available to you. Type:

```
IDL> LoadCT, 2
IDL> zcolors = BytScl(z, Top=!D.Table_Size-1)
```

The *Circle* program you are using in this example has a couple of weaknesses. Its major deficiency is that it doesn't always produce circles! If you specify the coordinates of the circle in the data coordinate system, the circle may show up on the display as an ellipse, depending upon the aspect ratio of the plot and other factors. (To obtain an excellent circle program, download the program *TVCircle* from the NASA Goddard Astrophysics IDL Library. You can find the library with a web browser at this World Wide Web address: <http://idlastro.gsfc.nasa.gov/homepage.html>.)

To avoid this deficiency in the *Circle* program, convert the data coordinates to device coordinates with the *Convert_Coord* command. Type:

```
IDL> coords = Convert_Coord(x, y, /Data, /To_Device)
IDL> x = coords(0, *)
IDL> y = coords(1, *)
```

Now, you are finally ready to use the *Polyfill* command to draw the colored circles that represent the data Z values. Type:

```
IDL> For j=0, 29 DO Polyfill, Circle(x(j), y(j), 10), $
      /Fill, Color=zcolors(j), /Device
```

As an added touch, it would be nice to have a color bar that can tell you something about the Z values and what the color means. You can add a color bar to the plot with the *Colorbar* program that came with this book. Type:

```
IDL> Colorbar, Position = [0.85, 0.15, 0.90, 0.95], $
      Range=[Min(z),Max(z)], /Vertical, $
      Format='(I5)', Title='Z Values'
```

Your output should look similar to the illustration in Figure 32.

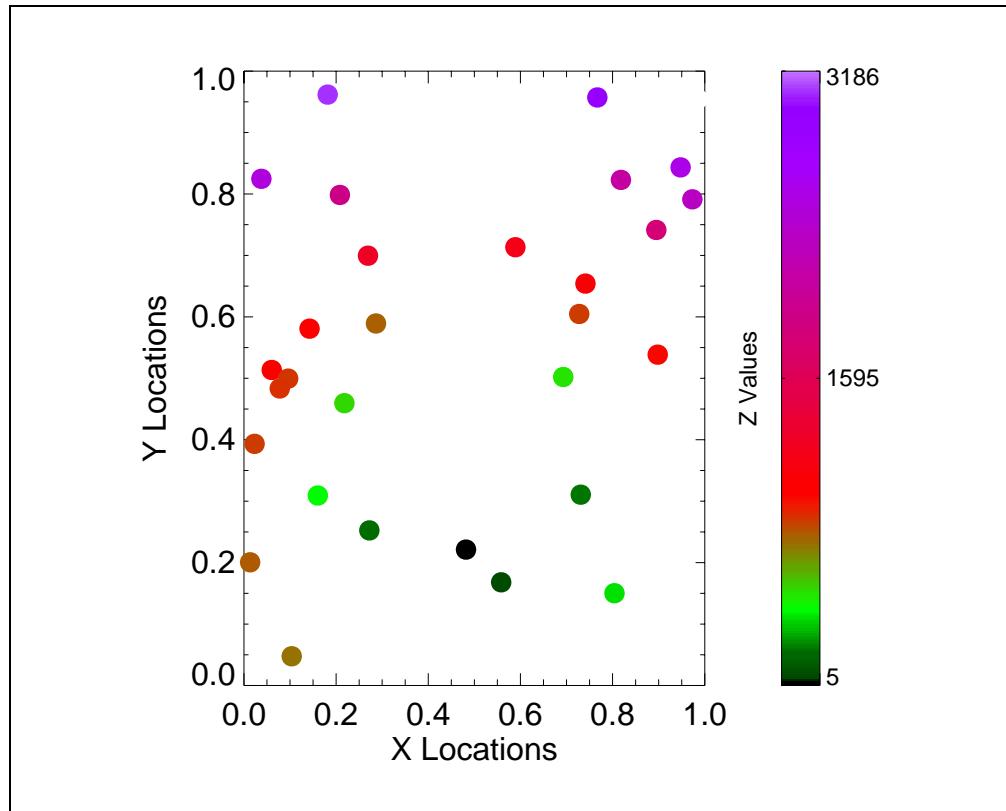


Figure 32: The color of the circles represents a third dimension in this 2D plot.

Chapter 3

◆ Discovering the Possibilities ◆◆◆



Working with Image Data

Chapter Overview

IDL got its start as a language for handling and processing images. It is no surprise that many scientists and engineers the world over still use it for that reason. This chapter lays the groundwork for working with images. Among the topics you will learn about are these:

- How to read and display image data
- The difference between 8-bit and 24-bit images
- How to scale image data
- How to position an image in the display window
- How to change image sizes
- How to read images from the display device
- How to perform basic image processing tasks
- How to construct simple image filters

Working with Images

In reality, any 2D data set of any data type can be thought of as an “image.” But to display image data on an 8-bit display, the image data must be “scaled” into byte values between 0 and 255. (On a 24-bit display, each of the RGB values that make up the 24-bit image must be scaled into byte values.) Since images are always displayed as byte values, they are often stored that way too. But, no matter how they are stored (and many images these days are stored as 16-bit integers, 32-bit integers, and even floating point values), they are always displayed as byte values with one or the other of the two IDL commands that display images: *TV* and *TVScl*.

To see how this works, you need some image data to work with. Use the *LoadData* command to open the *Ali and Dave* image data set. You will want to work with the second image in this two image data set. Type:

```
IDL> image = LoadData(10)
IDL> image = image[*, *, 1]
```

Displaying Images

You can use either of two IDL commands to display your image: *TV* or *TVScl*. These commands are identical in almost every way, including the keywords that can be used with them. They differ in only one respect: *TVScl* scales the image data into byte values that correspond to the number of colors you are using in your IDL session. (On 24-bit displays, the image data is scaled into the range 0 to 255.) For example, if you are using 220 colors in your IDL session, *TVScl* scales the image data into byte values between 0 and 219 before the image is displayed.

The *TV* command, on the other hand, takes the image data at face value and simply transfers it to the display as byte data. Image data values are truncated to byte values if necessary. If the image data is not scaled into the range of 0 to 255, there is an excellent chance the image will be displayed incorrectly.



Notice that unlike the *Plot*, *Surface*, and *Contour* commands, the *TV* and *TVScl* commands do not erase the display before displaying the image. Most of the time this is not a problem, but sometimes it is. If you want a clear display window to view your image data, you can erase whatever is currently displayed in the window with a simple *Erase* command, like this:

```
IDL> Erase
```

Here is an example to illustrate the point. The image data set you just read into IDL has already been scaled into the range of 0 to 255. You can see this is so by typing:

```
IDL> Print, Max(image), Min(image)
```

But, if you are working on an 8-bit display, chances are very good that you are not using all 256 colors that are available on your display device. To see how many colors you *are* using, type:

```
IDL> Print, !D.Table_Size
```

The number of colors used in an IDL session on an 8-bit display (here represented by the size of the color table) is usually in the range of 210-240 colors, but it can be considerably less. On a 24-bit display, you will have access to 16.7 million colors, but your color table size will still be just 256. You will learn later how IDL selects the number of colors it uses.

If you are running IDL on an 8-bit display, open a window, load the gray-scale color table, and display the image with the *TV* command, like this:

```
IDL> Window, 0, XSize=192, YSize=192
IDL> LoadCT, 0
IDL> TV, image
```

Your output should look like the illustration in Figure 33.

Since you used the *TV* command, the data was transferred to the display without any scaling. Although it is not apparent yet, all the pixels with image values greater than the number of colors in the IDL session are set to the same color value. That is, pixels with values greater than *!D.Table_Size-1* are being displayed with the same color pixel. (In this case, you are seeing “colors” as shades of gray.)

You might be able to see the difference if you display the image with the *TVScl* command. (You won’t see any difference at all on a 24-bit display, since there you have all 256 possible byte values available to you.) Open another window and move it near the first. Use the *TVScl* command to display the image, like this:

```
IDL> Window, 1, XSize=192, YSize=192
IDL> TVScl, image
```



Figure 33: An image of David Stern, founder of Research Systems and creator of IDL. The other image in the people.dat data set is Ali Bahrami, the first employee of Research Systems. Both David and Ali are still involved in the development of IDL.

You may be able to see a difference in shading between the two images. Since this image data has a maximum value of only 238, the difference might be subtle. If you can't see the difference, try scaling the data between 0 and 255 first, like this:

```
IDL> WSet, 0
IDL> image = BytScl(image)
IDL> TV, image
IDL> WSet, 1
IDL> TVScl, image
```

If you still can't see the difference, try loading a color table. The Red Temperature color table might work. Type:

```
IDL> LoadCT, 3
```

Now, to see what *TVScl* does, scale the data and display it with the *TV* command:

```
IDL> Window, 2, XSize=192, YSize=192
IDL> scaled = BytScl(image, Top=!D.Table_Size-1)
IDL> TV, scaled
```

What you see in window 2 should be identical to what you see in window 1. This is what we mean when we say *TVScl* byte scales the data into the number of colors used in the IDL session.



Note that if the image in the display window is *not* displayed with a red-temperature color scale you may be running IDL on a 16- or 24-bit color display. If this is the case, be sure to turn color decomposition *off* for these exercises. (You will learn more about color decomposition in “Working with Colors in IDL” on page 81.) Type these commands:

```
IDL> Device, Decomposed=0
IDL> TV, scaled
```

If you are on a 16-bit or 24-bit display, you will need to re-issue each graphics command from now on after changing the color table in order to see the new colors take effect. On a 16-bit or 24-bit display, the colors in the color table are not directly indexed or linked to the colors on the display. Rather, the color table is used as a way for the image to find the color it should use for the pixel. But that pixel color is expressed directly.

In general, if you don't know whether your data is scaled or not, you probably want to use the *TVScl* command, since this will give your image the maximum possible contrast in pixel values. But if color is important to you (and it almost always is), then you probably *never* want to use the *TVScl* command. Instead, you will want to scale your image data yourself, and use the *TV* command to display it.

An Alternative Image Display Command

Although it almost embarrasses me to say it, I find that I seldom, if ever, use either *TV* or *TVScl* to display image data in the real world. They just contain too many limitations to work well on different computers, with different graphics cards, etc. You will learn what I mean by limitations as you work through this chapter. But to be used correctly and generally, they have to be proceeded by 10-12 additional commands to check for such things as the type of image, the depth of the graphical display, the color decomposition state, etc. As always in IDL, when you find yourself typing the same 10 commands over and over again, you soon think to write an IDL program to do the work for you.

I've written such a program and named it *TVImage*. It is among the programs you downloaded to use with this book. You will learn of its advantages as you work though the examples in this chapter. Of course, in the end *TVImage* displays the image with the *TV* command, as it must. But it does a lot of work for you before you get to that part of the code. One of the things it does is relieves you of having to know the color decomposition state of your graphics device. So, you would not have to issue the *Device, Decomposed=0* command above to display an image. You could just do so directly with *TVImage*:

```
IDL> TVImage, scaled
```

Other commands like *TVImage* exist in the IDL community. One I like very much is a program named *IMDisp*, written by Liam Gumley and available from Liam's web page (<http://cimss.ssec.wisc.edu/~gumley/index.html>). I encourage you to look at both of these programs if you are having trouble displaying images properly in your IDL code.

Scaling Image Data

Suppose you are measuring atmospheric pressures and displaying the data as an image next to a color bar. You probably want to be sure that the image data you collected this week could be compared with image data you collected last week. In other words, you want to be sure a particular color, say red, in *this* data set indicates the same pressure as the red color in *that* data set last week.

If you display both this week's and last week's image data with the *TVScl* command, you have absolutely no guarantee that the particular red color means the same thing in both data sets.

The discrepancies arise from two sources. First, you may not be using the same number of colors in this IDL session today that you were using in your IDL session last week. Since *TVScl* scales the image data into the number of colors in your IDL session, this could introduce errors. Second, you have no guarantee that your data sets have the same range of data in them. Thus, the scaling with *TVScl* could have, again, introduced errors.

To circumvent these problems you want to scale the data sets with the *BytScl* command and display them with the *TV* command. To make certain that the number of colors in your IDL session will not introduce errors, you want to scale the data into the same number of "bins". And, to make certain the range of data in the data sets doesn't introduce errors, you want to scale the data into the same data range.

You do this by applying the keywords *Top*, *Min*, and *Max* to the *BytScl* command. For example, suppose you always want to display your data in 100 different shades of gray or colors. And suppose that the minimum data value you expect in any data set is 15, whereas the maximum valid data value is 245. Then the *BytScl* command is used like this:

```
IDL> scaledImage = BytScl(image, Min=15, Max=245, Top=99)
```

In this example, any value in the data set that has a value less than 15 will be set to the value of 15 before the data is scaled. Similarly, any value in the data set with a value greater than 245 will be set to 245 before the data is scaled.

Once your data is scaled, it is displayed with the *TV* command, like this:

```
IDL> TV, scaledImage
```

Now, if you always scale your data sets in the same way (and you always have at least 100 shades of gray or colors in your IDL session), your data set from last week can be compared directly with this week's data set. A particular color red will always indicate a *specific* data range or pressure.

At this point, you may have a number of graphics windows on the display. Here is a trick to get rid of all the open windows in a single command. Type:

```
IDL> WHILE !D.Window NE -1 DO WDelete, !D.Window
```

Scaling Images into Different Portions of the Color Table

Another reason to know how to scale image data, is to be able to scale your data into different portions of the color table when using 8-bit display devices. This makes it possible for images to appear to be displayed with different color tables, or for you to reserve specific portions of the color table for specific purposes. For example, you may want to have a portion of the color table reserved for graphics drawing colors.



Note that one of the huge advantages of using 24-bit color displays is that you can use an unlimited number of color tables all at once. The downside of 24-bit displays is that you have to re-issue the graphics command (e.g., the *TV* command) after you change the color table in order to see the new colors take effect. You will see how to write programs that can automatically re-issue the graphics command when a new color table is loaded later in the book.

There is just one physical color table on most 8-bit computers, and it is used in all IDL graphics windows. But by manipulating this color table you can make it appear that you have several different color tables loaded simultaneously. You do this by loading different color tables into different portions of the one physical color table. Perhaps the easiest way to do this is to use the *NColors* and *Bottom* keywords to the *LoadCT* or *XLoadCT* commands.

For example, suppose you want to display the same image in what appears to be two different color tables. After you have opened a graphics window in IDL, you can tell how many colors are in the color table in that IDL session by examining the system variable *!D.Table_Size*. If you divide this number by two, you know how many colors to use for each image:

```
IDL> half = !D.Table_Size / 2
```

To display the image data *image* in the same window in what appears to be two different color tables, you must scale the image data into these two portions of the color space. First, use the *BytScl* command to scale the image data into the first portion of the color table. Make a new image, *image1*, like this:

```
IDL> image1 = BytScl(image, Top=half-1)
```

Now, make a second image, *image2*, by scaling the image data into the second portion of the color table, like this:

```
IDL> image2 = ByteScl(image, Top=half-1) + Byte(half)
```

You want the image on the left to be displayed with a gray-scale color table (table number 0). To do so, you must load those gray-scale colors in the portion of the color table occupied by the first image's data values. Type:

```
IDL> LoadCT, 0, NColors=half, Bottom=0
```

You can interactively choose any color table you like for the image on the right if you use the *XLoadCT* command to load those colors in just the second portion of the color table. Like this:

```
IDL> XLoadCT, NColors=half, Bottom=half
```

Finally, put the two scaled images side-by-side into the same window, like this. Notice that you are using the *TV* command. Do you understand why?

```
IDL> Window, XSize=192*2, YSize=192
IDL> Device, Decomposed=0
IDL> TV, image1
IDL> TV, image2, 192, 0
```

To restore a single normal color table before continuing with the examples in this chapter, type:

```
IDL> LoadCT, 0
```

Displaying 8-Bit Images with Different Color Tables on 24-Bit Displays

Using different color tables when you are running on a 16-bit or 24-bit display device is as simple as loading different color tables and displaying the image. For example, if you are running on a 16-bit or 24-bit device, try this:

```
IDL> world = LoadData(7)
IDL> Window, 1, Title='Gray Scale Image'
IDL> LoadCT, 0
IDL> TV, world
IDL> Window, 2, Title='Color Image'
IDL> LoadCT, 5
IDL> TV, world
```

If both of these images were displayed in gray-scale colors, then you are certainly running IDL on a 16-bit or 24-bit display with color decomposition on. You will learn more about color decomposition later, but for now make sure color decomposition is off. Type this command, then recall the commands above to get more colorful results.

```
IDL> Device, Decomposed=0
```

Displaying 24-Bit Images

True color, or 24-bit, images can also be displayed with the *TV* command. A 24-bit image is always a three-dimensional data set, with one of its dimensions set to 3. For example, the data set could be an *m*-by-*n*-by-3 array, in which case the image is said to be *band-interleaved*. If the image is *m*-by-3-by-*n* it is said to be *row-interleaved*; and if it is 3-by-*m*-by-*n* it is said to be *pixel-interleaved*.

To load a 24-bit image, type this command:

```
IDL> rose = LoadData(16)
```

This data set is a pixel-interleaved image. You can see this by typing this command:

```
IDL> Help, rose
ROSE           BYTE      = Array [3, 227, 149]
```

To display a 24-bit image on either an 8-bit or 24-bit display, simply use the *True* keyword to indicate what kind of interleaving is being used. *True=1* means pixel-interleaved; *True=2* means row-interleaved; and *True=3* means band-interleaved.

```
IDL> Window, XSize=227, YSize=149
IDL> TV, rose, True=1 ; Pixel-interleaved
```

The output should look similar to that in Figure 34 and in color if you are on a 24-bit display. It should appear in gray-scale on an 8-bit display. If that isn't what you expected, please read on.



Figure 34: The rose data set as it is supposed to look. If your output doesn't look like a beautiful rose, try turning decomposed color on and display it again.

If your output doesn't look like a beautiful rose, the problem may be that you are running IDL on a PC or Macintosh computer, with 24-bit color turned on, and with color decomposition turned off. (UNIX computers will display a 24-bit image in the correct colors, no matter what the color decomposition setting, but this is not true for other computers.) Set the color decomposition keyword and re-display the image:

```
IDL> Device, Decomposed=1
IDL> TV, rose, True=1
```



Unfortunately, 24-bit images always appear in gray-scale on 8-bit display devices. To see the image in its actual colors on such devices, you have to create a 2D image and the red, green, and blue color tables to go with the image from the 24-bit or 3D image data set. This is done in IDL with the *Color_Quan* command. If you are on an 8-bit display device, type these commands:

```
IDL> image2d = Color_Quan(rose, 1, r, g, b)
IDL> TVLCT, r, g, b
IDL> TV, image2d
```

You will now see the image in color, although the color reproduction is never as good as the 24-bit image displayed on a 24-bit device.

Displaying 24-Bit Images on 24-Bit Displays

If you are on a 24-bit display, then the display of 24-bit images is just slightly more complicated. To correctly display a 24-bit image, color decomposition must be turned

on. This is done automatically on most workstations which are in 24-bit color, but is not done automatically in Windows or Macintosh 24-bit color. To be absolutely sure of seeing the 24-bit image in the correct image colors, you should type these commands on a 24-bit display:

```
IDL> Device, Decomposed=1  
IDL> TV, rose, True=1
```



Note that the *TVImage* program you downloaded to use with this book automatically sets the correct color decomposition mode and the correct *True* keyword, depending upon whether it is a 24-bit or 8-bit image you are displaying. It will also automatically apply the *Color_Quan* function if a 24-bit image is to be displayed on an 8-bit device. This is true even when the 8-bit device is the PostScript device.

```
IDL> TVImage, rose
```

Displaying 8-Bit Images on 24-Bit Displays

On a 24-bit display, 8-bit images are routed through the color table at the time they are displayed if color decomposition is turned off. (If color decomposition is turned on, then the 8-bit image is replicated in each image plane of the device and the result is a gray-scale image whether you have a color table loaded or not. Alas, not all versions of IDL work in exactly the same way. The situation I describe is the one in effect with IDL 5.3 at the time this book was written.) In other words, the pixel value of an 8-bit image is used as an index to look up the particular red, green, and blue color for that particular pixel. What this means is that if you change color tables in your IDL session, you must re-display the 2D image to see the new colors take effect. This is because color determination is made at the time the image is displayed on a 24-bit display and because you are using the RGB color model. And note especially that color decomposition must be turned *off*, or the color table vectors are ignored and the 8-bit image is *always* displayed in gray-scale colors. See “Loading Color Tables on a 24-Bit Display” on page 88 for more information. If you have a 24-bit display, type these commands:

```
IDL> world = LoadData(7)  
IDL> Window, XSize=360, YSize=360  
IDL> LoadCT, 5  
IDL> Device, Decomposed=0  
IDL> TV, world
```

To see the image in another color table, load the colors and re-issue the *TV* command to route the image pixel values through the color table vectors. Note that the image colors do *not* change when you issue the *LoadCT* command.

```
IDL> LoadCT, 3  
IDL> TV, world
```

Automatic Updating of Graphic Displays When Color Tables are Loaded

The color table changing tools *XLoadCT* (which is supplied with IDL) and *XColors* (which is supplied with the programs written for this book and which I prefer for many reasons to *XLoadCT*) both have the ability to call an IDL program when they load color table vectors in the hardware color table. For example, open a text editor and type the following short program. (You can use the editor built into the IDL Development Environment if you like. Choose *File->New->Editor* from the menu bar.)

```
PRO Display, Image=image, WID=wid  
IF N_Elements(wid) EQ 0 THEN wid = !D.Window  
WSet, wid  
TV, image  
END
```

Save the file as *display.pro* and compile the program code like this:

```
IDL> .Compile display
```

If the program has errors and doesn't compile for some reason, fix the errors and recompile.

Now, to see the image immediately update when the color table is changed, type this:

```
IDL> XColors, NotifyPro='Display', Image=world
```

Close the currently open *XColors* program.

You can use the *WID* keyword to select another window for the display. For example, you can type this:

```
IDL> Window, 5
IDL> image5 = LoadData(5)
IDL> TV, image5
IDL> XColors, NotifyPro='Display', Title='Window 5 Colors', $
      Image=image5, WID=5
IDL> Window, 4
IDL> image4 = LoadData(4)
IDL> TV, image4
IDL> XColors, NotifyPro='Display', Title='Window 4 Colors', $
      Image=image4, WID=4
```

Notice that your two *XColors* programs can exist on the display at the same time and that they display into the proper window. This is not possible with the RSI-supplied *XLoadCT*, which must limit itself to a single copy in order to protect its color tables with are in a common block. (You can have as many *XColors* programs as you like, as long as each has a different title.)

The equivalent capability for *XLoadCT* is located in the keywords *UpDateCallback* and *UpdateCbData*. The display program must be written so that it can accept one, and only one, keyword parameter, which must be named *Data*. For example, you could write the display procedure like this:

```
PRO XDisplay, Data=image
TV, image
END
```

Then call the *XLoadCT* program like this:

```
IDL> Window, 0
IDL> image = LoadData(7)
IDL> TV, image
IDL> XLoadCT, UpDateCallback='XDisplay', UpdateCbData=image
```

If you wanted to pass the window index number into the *XDisplay* program, you would have to make the *Data* variable into a structure. For example, like this:

```
PRO XDisplay, Data=struct
WSet, struct.wid
TV, struct.image
END
```

But this would mean you would *always* have to pass a window index number:

```
IDL> XLoadCT, UpDateCallback='XDisplay', $
      UpdateCbData={image:image, wid:!D.Window}
```

To my mind, this makes the *XLoadCT* method a lot less flexible than the *XColors* method for passing information back and forth. I find *XLoadCT* even less flexible in widget programs.

Controlling Image Display Order

Normally, when IDL displays an image, it uses the convention in which the 0th column and the 0th row of the image are at the lower-left corner of the image. Some people prefer to have the 0th column and 0th row be at the upper-left corner of the image. If you prefer the second convention, you can force IDL to use that convention by setting the system variable *!Order*. By default *!Order* is set to 0. If you would prefer all your images be displayed with the 0th column and 0th row at the upper-left corner of the image, set *!Order* = 1.

If you prefer that just the image you are displaying use the second convention, then use the *Order* keyword to either the *TV* or *TVScl* command. For example, to see the two conventions side-by-side, type:

```
IDL> Window, XSize=192*2, YSize=192
IDL> TVScl, image, Order=0
IDL> TVScl, image, Order=1, 192, 0
```



You might obtain an image data file from someone else and it will appear upside-down when it is displayed. This is most often because the person who created the data file used a different convention for placing the 0th column and row. Try reversing the *Order* keyword and see if this corrects the problem.

If you wish to fix the problem permanently and not bother with the *Order* keyword, you can easily reverse the order of the pixels in the Y (or second) dimension with the *Reverse* command, like this:

```
IDL> image = Reverse(image, 2)
```

Changing Image Size

IDL provides two commands to change an image's size: *Rebin* and *Congrid*.

Rebin is limited in that the newly-created image must have a size that is an integral multiple or factor of the original image. For example, your variable *image* may be resized to 192/2 or 192*3 elements in either the X or Y direction, but not to 300 or 500 elements. The image may be reduced in size in one direction and increased in size in the other dimension. For example, to resize the variable *image* to 384 columns and 96 rows, type this code. Your output will look similar to the illustration in Figure 35.

```
IDL> Window, XSize=384, YSize=96
IDL> new = Rebin(image, 384, 96)
IDL> TVScl, new
```



Figure 35: Images resized with the *Rebin* command must be integer multiples of the original image size.

By default, *Rebin* uses bilinear interpolation when magnifying an image, and nearest neighbor averaging when reducing an image. Nearest neighbor sampling can be used in both directions if the *Sample* keyword is set. Bilinear interpolation is more accurate, but requires more computer time.

```
IDL> Window, XSize=192/2, YSize=192/2
IDL> new = Rebin(image, 96, 96, /Sample)
IDL> TVScl, new
```

Congrid is similar to *Rebin*, except in two ways. First, the number of columns and rows specified in the new image may be set to any arbitrary number. And second, nearest neighbor sampling is used by default. If you want to use bilinear interpolation, you must set the *Interp* keyword, like this:

```
IDL> Window, XSize=600, YSize=400
IDL> new = Congrid(image, 600, 400, /Interp)
IDL> TVScl, new
```

Changing Image Size in PostScript

Devices, like the PostScript or Printer device, that have scalable pixels (as opposed to the fixed-size pixels of your display device) size their images differently. (See “Differences Between the Display and PostScript Devices” on page 185 for more detailed information.) In particular, you don’t use the *Rebin* or *Congrid* commands to size images, you use the *XSize* and *YSize* keywords to the *TV* or *TVScl* command instead.

For example, instead of resizing the image above to a 600 by 400 size with the *Congrid* command, if you wanted the image to have an aspect ratio of 6 to 4, you might do this when you output into a PostScript file:

```
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PS'
IDL> Device, XSize=6, YSize=4, /Inches
IDL> TVScl, image, XSize=6, YSize=4, /Inches
IDL> Set_Plot, this Device
```

If images are sized and positioned with normalized coordinates as described in the next section, you can write image display code that is essentially independent of the graphics display device.



Note that in addition to keeping track of whether you are trying to display an 8-bit or a 24-bit image, the current decomposed state, the required decomposed state, the depth of the output display device, and whether you are displaying multiple images in the same window, *TVImage* is also smart enough to know *where* you are trying to display the image and can adjust itself accordingly if you are sending output to a PostScript file or to a printer. Just another reason why I couldn’t possibly live without it. Read on for even more reasons!

Positioning an Image in the Display Window

Normally when an image is displayed, IDL puts the lower-left corner of the image in the lower-left corner of the window. But you can move the image to other positions in the display window by means of additional parameters to the *TV* and *TVScl* commands.

For example, if a second parameter is present, this is interpreted as the *position* of the image in the window. The *position* is calculated from the size of the window and the size of the image. See the on-line help for the *TV* command for the detailed algorithm. Type:

```
IDL> ? TV
```

The positions start at the top-left of the display and continue to the bottom-right. For example, in a 384 by 384 window, there are four positions for a 192 by 192 image, starting in the upper-left corner of the window. Try typing the following commands:

```
IDL> Window, XSize=384, YSize=384
```

```
IDL> TVScl, image, 0
IDL> TVScl, image, 1
IDL> TVScl, image, 2
IDL> TVScl, image, 3
```

It is also possible to position an image in a window by specifying the pixel location of the lower-left hand corner of the image explicitly. This is done by specifying two additional parameters to the *TV* or *TvScl* commands after the name of the image data. For example, to locate the 192 by 192 image named *image* in the middle of the window you just created, type:

```
IDL> Erase, Color=!D.Table_Size-1
IDL> TVScl, image, 96, 96
```

In this case, you placed the lower-left corner of the image at pixel location (96, 96). Positioning an image in this way is important when you want to leave room for such additional graphic elements as color bars or other annotations.



For example, type these commands to display a color bar on the left-hand side and the image on the right-hand side of a window. Your display window should look similar to the illustration in Figure 36.



Figure 36: The image with a color bar next to it showing the color gradations.

```
IDL> Window, XSize=320, YSize=320
IDL> ncolors = !D.Table_Size
IDL> TVLCT, 255, 255, 0, ncolors-1
IDL> Erase, color=ncolors-1
IDL> colorbar = Replicate(1B,20) # BIndGen(256)
IDL> TV, BytScl(colorbar, Top=ncolors-2), 32, 36
IDL> TV, BytScl(image, Top=ncolors-2), 92, 64
```

Positioning Images with Normalized Coordinates

It is frequently convenient to position and size images using normalized coordinates. This is similar to how the *Position* keyword is used with other IDL graphics commands. (See “Differences Between the Display and PostScript Devices” on page 185 for additional information.) This is especially true if you are displaying images in resizeable graphics windows, or using images with other IDL graphics

routines in the same window, or you want to write IDL programs that can be sent to a PostScript file with little or no trouble. This last point is especially pertinent.

Take, for example, the IDL commands you just typed above to put a color bar next to an image. While these commands work well in a display window, they don't work at all if you want similar PostScript output. (See "Differences Between the Display and PostScript Devices" on page 185 for a full account of the problems you might face.)

But suppose for a moment you could specify the size and location of an image in the window with a *Position* keyword. How would this work? Assume you want to put the image in a window, *any* window, and have it fill up, say 80% of the window space. (It has been pointed out to me that my formulation below uses only 64% of the window space, but the truth seems so non-intuitive I am loath to correct the mistake. Make it a "large percent" of the window space, if you like). In terms of normalized coordinates, you can say the position in the window can be expressed like this:

```
position = [0.1, 0.1, 0.9, 0.9]
```

But how does this translate to device coordinates, which are normally used for images? It depends, naturally, on how big the window is. But you can find out how big the visible portion of the display window is. It is given by the system variables *!D.X_VSize* and *!D.Y_VSize*, expressed in device or pixel units.

So, in terms of pixel coordinates, you can calculate the size the image has to be and where it has to start in the output window like this:

```
xsize = (position[2] - position[0]) * !D.X_VSize
ysize = (position[3] - position[1]) * !D.Y_VSize
xstart = position[0] * !D.X_VSize
ystart = position[1] * !D.Y_VSize
```

The only difference between outputting the image to the display device and outputting the image to a PostScript file will be how it is sized. You can write the code to display the image like this:

```
IF !D.Name EQ 'PS' THEN $
    TV, image, XSize=xsize, YSize=ysize, xstart, ystart $
ELSE $
    TV, Congrid(image, xsize, ysize), xstart, ystart
```

This bit of code will work whether you are outputting the image to the display or to a PostScript file. But the aspect ratio of the image is not always maintained when you do this. In fact, you allow the image to fit the shape of the window. This works well for some applications and not so well for others. In any case, the problem is easily solved because if you want to keep track of and preserve the aspect ratio of the image you simply fit one side of the image and adjust the position coordinates of the other appropriately.

The code to do this has already been written for you in the program *TVImage*, which you downloaded to use with this book. *TVImage* uses the *Position* keyword to position and size images for display, either on your display terminal or in a PostScript file. You can use the *Keep_Aspect_Ratio* keyword if you want the *TVImage* program to maintain the aspect ratio of the image it is displaying.

You can reproduce the display above in which the color bar was to the left of the image using *TVImage*, like this:

```
IDL> Erase, color=ncolors-1
IDL> barPosition = [32, 32, 52, 292]/320.0
IDL> imagePosition = [92, 64, 284, 256]/320.0
IDL> colorbar = Replicate(1B,20) # BIndGen(256)
IDL> TVImage, BytScl(colorbar, Top=ncolors-2), $
    Position=barPosition
```

```
IDL> TVImage, BytScl(image, Top=ncolors-2), $  
Position=imagePosition
```

This is not only much better because it can go into any size window or into a PostScript file as well as to your display, but it also opens up the possibility of easily adding additional graphics to this display. For example, here is how easy it is to put a box and labels around the color bar and the image.

```
IDL> TVLCT, 255, 255, 255, ncolors-1  
IDL> Plot, [0, !D.Table_Size], YRange=[0,!D.Table_Size], $  
/NoData, Color=0, Position=barPosition, XTicks=1, $  
/NoErase, XStyle=1, YStyle=1, XTickFormat='(A1)', $  
YTicks=4  
IDL> Plot, IndGen(192), IndGen(192), /NoData, $  
Position=imagePosition, /NoErase, $  
XStyle=1, YStyle=1, Color=0
```

Your output should look like the illustration in Figure 37.

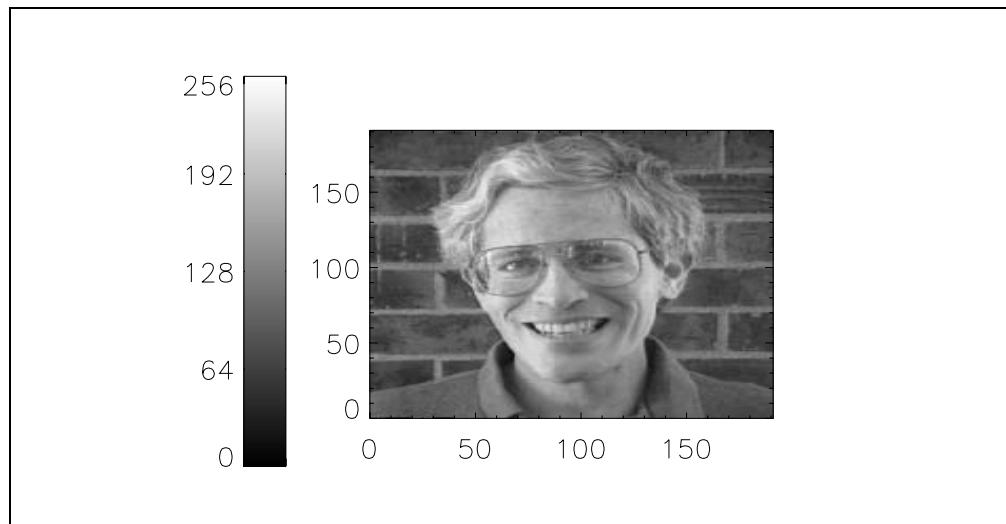


Figure 37: Using the *TVImage* command not only allows you to position images in a device-independent way, it also makes it easy to use other graphics commands.

Reading Images from the Display Device

There are times when you spend a lot of time and issue a lot of commands to get a graphic display just the way you like it. And sometimes it is convenient to read the graphic display into an image variable for manipulation or even hardcopy output. This is the time when you need to know how to get a screen capture of an IDL graphics window. You can do this with the *TVRD* command, which reads the contents of an IDL graphics window into either a 2D or 3D IDL byte array, depending upon the depth of your display device.

Obtaining Screen Dumps on 8-Bit Displays

To read the entire graphics window on an 8-bit display, type these commands:

```
IDL> Window, XSize=250, YSize=250  
IDL> TVScl, image  
IDL> new_image = TVRD()  
IDL> Help, new_image
```

Notice that the newly created variable is now a 250 by 250 byte array.

Obtaining Screen Dumps on 24-Bit Displays

If you are running IDL on a 16- or 24-bit display, you don't want to use the *TVRD* command like shown above. A 16- or 24-bit display has three color channels. If you use the *TVRD* command without any arguments, as above, then what you get in the 2D resulting array is the maximum pixel value in each of those three channels. Unless you have a gray-scale color table loaded (in which case, each channel has the same value), this is not what you want or expect.

Moreover, when you do a screen dump on a 24-bit display on a PC or Macintosh computer, if you have color decomposition turned off, the numbers you read from the display are back-translated through the color table. (Who thinks this stuff up!?) Thus, it is essential to make sure you have color decomposition turned on before you take the screen dump. The commands you want to use are these:

```
IDL> Device, Decomposed=1
IDL> new_image = TVRD(True=1)
```

But now notice that you have a 24-bit image, not a 2D 8-bit image, and you will have to use the appropriate *True* keyword with the *TV* command when you display the image:

```
IDL> Help, new_image
IDL> Erase
IDL> Device, Decomposed=1
IDL> TV, new_image, True=1
```



Note that neither the *Device* command nor the *True* keyword is necessary with *TVImage*, since the program automatically determines whether it is displaying an 8-bit or 24-bit image and sets the correct color decomposition value and *True* keyword.

```
IDL> TVImage, new_image
```

Reading a Portion of the Display

If you want to read just a portion of the window, you can specify the pixel coordinates of the lower left corner of the part of the window you want and the number of columns and rows to read. In other words, you can specify a rectangle. For example, if you wanted to capture just the face in the image above, and you are on an 8-bit display device, you can type this:

```
IDL> new_image = TVRD(40, 30, 110, 130)
```

Or, if you are on a 24-bit display device, you can type this:

```
IDL> Device, Decomposed=1
IDL> new_image = TVRD(40, 30, 110, 130, True=1)
```

The image array you obtain from reading the display device can be treated like any other image in IDL. For example, if the screen dump is an 8-bit image, you display it like this:

```
IDL> Erase
IDL> TV, new_image
```

Or, if the screen dump is a 24-bit image, you display it like this:

```
IDL> Device, Decomposed=1
IDL> TV, new_image, True=1
```

An Alternative to TVRD

A color decomposition independent alternative to *TVRD*, named *TVREAD*, is available among the programs you downloaded to use with this book. It can be used in exactly the same way *TVRD* is used, but you don't have to be aware of the state of the

Decomposed keyword to the *Device* command. Nor do you have to worry about setting the *True* keyword in *TVREAD*, as you do in *TVRD*. The return image will be an 8-bit image if the screen dump is done on an 8-bit display, and a 24-bit image if the screen dump is done on a 24-bit display.

Another advantage of *TVREAD* is that it can automatically send the screen dump to a BMP, GIF, JPEG, PICT, PNG, or TIFF file. You don't have to worry about the depth of your display device, or the state of color decomposition. For example, to display two images one above the other in a window and create a JPEG file of result, type these commands:

```
IDL> LoadCT, 4
IDL> Window, XSize=350, YSize=500
IDL> !P.Multi = [0, 1, 2]
IDL> TVImage, LoadData(5)
IDL> TVImage, LoadData(7)
IDL> image = TVREAD(/JPEG)
IDL> !P.Multi = 0
```

Basic Image Processing in IDL

IDL originated as an image processing tool, so it has many image processing capabilities. What are described in this section are a few of the basic image processing tools available in IDL.

Histogram Equalization

If you look at the distribution of pixel values in an image, you will often find that the distribution tends to cluster in a narrow range of values. In effect, the image has a very narrow dynamic color range. If you spread the pixel distribution out, so that each subrange of values had approximately the same number of pixels with those values, the information content of the image can sometimes be improved. This process of spreading the pixel distribution over the entire dynamic color range is known as histogram equalization.

For example, open the data set *CT Scan Thoracic Cavity* with the *LoadData* command. This image is a CT scan with a narrow dynamic color range.

```
IDL> scan = LoadData(5)
```

To see a histogram plot of the pixel value distribution of the variable *scan*, type these commands. Your graphics window will look similar to the illustration in Figure 38.

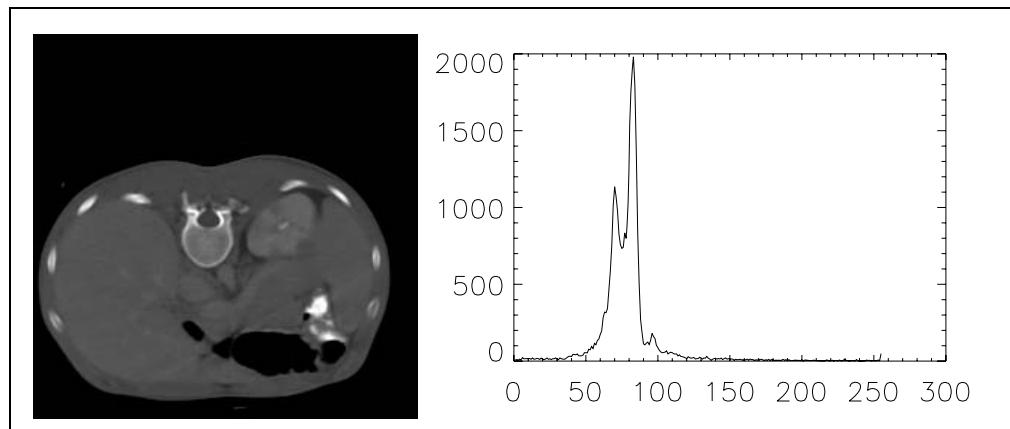


Figure 38: A normal image with a narrow pixel distribution. Here most of the pixels have values between 50 and 100.

```
IDL> LoadCT, 0
IDL> Window, 0, XSize=600, YSize=250
IDL> TV, scan
IDL> Plot, Histogram(scan), /NoErase, Max_Value=5000, $
    Position=[0.5, 0.15, 0.95, 0.95]
```

You see that most of the pixels have values that fall between 50 and 100. To spread the pixel distribution out over the whole color range, so that there are approximately equal number of pixels with each possible color value, use the *Hist_Equal* command, like this:

```
IDL> equalized = Hist_Equal(scan)
```

To see the new pixel distribution histogram and the histogram-equalized image, type:

```
IDL> Window, 1, XSize=600, YSize=250
IDL> TV, equalized
IDL> Plot, Histogram(equalized), Max_Value=5000, $
    Position=[0.5, 0.15, 0.95, 0.95], /NoErase
```

The histogram equalized image should look similar to the illustration in Figure 39.

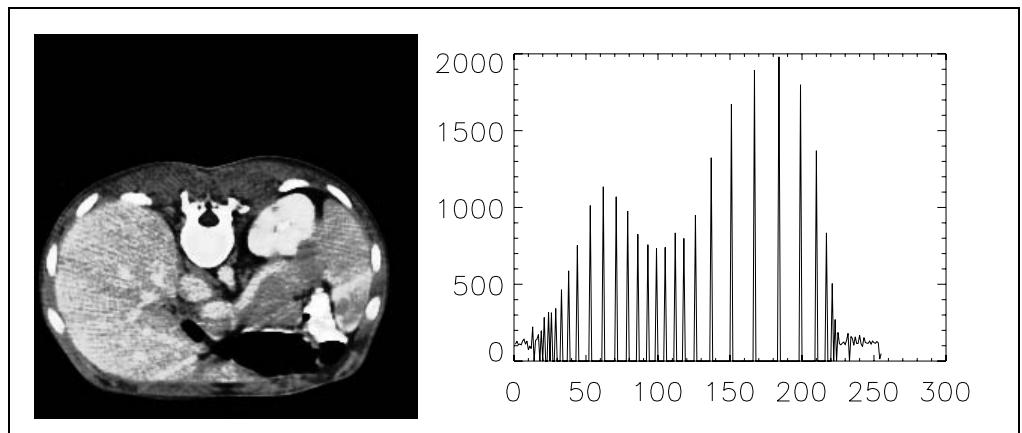


Figure 39: A histogram-equalized image. The pixel distribution has been spread out over the entire color dynamic range.

Smoothing Images

Images can be smoothed by averaging each pixel value with the values of its surrounding neighbors. This is known as mean or boxcar smoothing. Mean smoothing is accomplished in IDL with the *Smooth* function, which performs an equally weighted smoothing using a square neighborhood of a given odd width. For example, if the neighborhood is 3-by-3, then each pixel is replaced by the mean value of it and its eight surrounding pixels.

To see an unsmoothed image and the image smoothed with a 5-by-5 boxcar average, type:

```
IDL> Window, 0, XSize=192*3, YSize=192
IDL> TV, image, 0, 0
IDL> smoothed = Smooth(image, 5, /Edge_Truncate)
IDL> TV, smoothed, 192, 0
```

Notice the *Edge_Truncate* keyword used with the *Smooth* command. This keyword replicates pixels near the edge of the image so that a smoothing occurs over the entire image. If the keyword is not used, pixels near the edge of the image are simply replicated and not smoothed.

Image smoothing is used in an image processing technique called *unsharp masking*. This is a technique that is used to locate edges in images, or those locations where pixel values change dramatically. The technique is quite simple: subtract the smoothed image from the unsmoothed image, like this:

```
IDL> TV, ((image - smoothed) + 255) / 2.0, 2*192, 0
```

Your display window should look similar to the illustration in Figure 40.



Figure 40: The original image on the left, the smoothed image in the middle, and the unsharp masked image on the right.

Using the *Smooth* command, you are giving equal weight to the neighboring pixels to determine the mean value. This sometimes results in unwanted blurring of the image. Another approach is to use a process called *convolution* to do the smoothing. In this technique, a square kernel is convolved with the image. For example, the *Smooth* command, in its 3-by-3 case, uses a kernel like this:

```
1   1   1
1   1   1
1   1   1
```

You will get less blurring in your image if you give more weight to the central pixel and less to its neighbors. For example, you might want to create a kernel like this:

```
1   2   1
2   8   2
1   2   1
```

To convolve the image with this kernel, use the *Convol* command, like this:

```
IDL> kernel = [[1,2,1], [2,8,2], [1,2,1]]
IDL> TV, image, 0, 0
IDL> TV, Smooth(image, 3, /Edge_Truncate), 192, 0
IDL> TV, Convol(image, kernel, Total(kernel), $
    /Edge_Truncate), 2*192, 0
```

You can, of course, create kernels of any size. Here is a typical Gaussian distribution 5-by-5 kernel:

```
1   2   3   2   1
2   7   11  7   2
3   11  17  11  3
2   7   11  7   2
1   2   3   2   1
```

This can be created and applied to the image like this:

```
IDL> kernel = [[1,2,3,2,1], [2,7,11,7,2], $ [3,11,17,11,3],
    [2,7,11,7,2], [1,2,3,2,1]]
```

```
IDL> TV, Convol(image, kernel, Total(kernel), $  
/Edge_Truncate), 192*2, 0
```

You can find descriptions of image kernels in almost any good image processing book.

Removing Noise From Images

A common image processing technique is to remove noise from an image. Noise can accumulate from many sources and often acts to degrade the image. A common form of noise is salt and pepper noise, in which random pixels in the image have extreme values. To see how image smoothing deals with this kind of noise, first create a noisy image. Use the image from before, and type the commands below to turn approximately 10 percent of its pixels into salt and pepper noise:

```
IDL> noisy = image  
IDL> points = RandomU(seed, 1800) * 192 * 192  
IDL> noisy(points) = 255  
IDL> points = RandomU(seed, 1800) * 192 * 192  
IDL> noisy(points) = 0
```

Create a window and display the noisy image next to the original, like this:

```
IDL> Window, XSize=192*3, YSize=192  
IDL> TV, image, 0, 0  
IDL> TV, noisy, 192, 0
```

The *Median* command in IDL is an excellent choice to remove salt and pepper noise from images. The *Median* command is similar to the *Smooth* command, except that it calculates the median value of the pixel neighborhood instead of the mean value. This has two important effects. First, it can eliminate extreme values in images. And second, it does not blur the edges or features of images whose sizes are larger than the neighborhood. To see how this works, type:

```
IDL> TV, Median(noisy, 3), 2*192, 0
```

Your display window should look similar to the illustration in Figure 41.



Figure 41: The original image on the left, the noisy image in the center, and the noisy image smoothed with a median filter on the right.

Enhancing the Edges of Images

An image may be sharpened or have its edges enhanced by differentiation. IDL provides two built-in edge enhancement functions, *Roberts* and *Sobel*, and there are other ways to enhance image edges as well. For example, you can convolve the image with a Laplacian kernel that looks like this:

```
1   1   1  
1  -7   1  
1   1   1
```

This is often called a Laplacian sharpening operator because it improves contrast at the edges of images. To see how these methods work, type:

```
IDL> TV, Sobel(image), 0  
IDL> TV, Roberts(image), 1  
IDL> kernel = [[1,1,1], [1,-7,1], [1,1,1]]  
IDL> TV, Convol(image, kernel), 2
```

Your display window should look similar to the illustration in Figure 42, below.



Figure 42: Three different ways to enhance the edges of images. On the left, the Sobel method. The Roberts method is shown in the center. Convolving the image with a Laplacian kernel is shown on the right.

Frequency Domain Filtering of Images

Filtering in the frequency domain is a common image and signal processing technique. It can be used to smooth, sharpen, de-blur, and restore images.

There are three basic steps to frequency domain filtering:

1. The image is transformed from the spatial domain into the frequency domain using the Fast Fourier Transform (FFT).
2. The transformed image is multiplied by a frequency filter.
3. The filtered image is transformed back to the spatial domain.

These steps are implemented in IDL with the Fast Fourier Transform function *FFT*. (The image is transformed from the spatial domain to the frequency domain if the second positional parameter to the FFT command is a negative 1. The transformation occurs in the opposite direction if the parameter is a positive 1.) The general form of the frequency filtering command is this:

```
filtered_image = FFT(FFT(image, -1)*filter, 1)
```

In this case, *image* is either a vector or a two-dimensional image and *filter* is a vector or two-dimensional array designed to filter out certain frequencies in the image.

Building Image Filters

IDL makes it easy to build digital image filters with its array-oriented operators and functions. (In fact, many programmers take advantage of the *Digital_Filter* command that comes with IDL to build digital filters.) Many common filters take advantage of what is called a frequency image or Euclidean distance map. An Euclidean distance map for a two-dimensional image is an array of the same size as the image. Each pixel in the distance map is given a value that is equal to its distance from the nearest corner of the two-dimensional array. The *Dist* command in IDL is used to create the

Euclidean distance map or frequency image. To see a representation of a simple distance map, type:

```
IDL> Surface, Dist(40)
```

A common type of filter to use in frequency domain filtering is a Butterworth frequency filter. The general form of a low-pass Butterworth frequency filter is given by the equation:

$$\text{filter} = 1 / [1 + C(R/R_o)^{2n}]$$

where the constant C is set equal to 1.0 or 0.414 [the value defines the magnitude of the filter at the point where $R=R_o$ as either 50% or $1/\sqrt{2}$], R is the frequency image, R_o is the nominal filter cutoff frequency (represented as a pixel width in practice), and n is the order of the filter and is usually 1.

A high-pass Butterworth frequency filter is given by the equation:

$$\text{filter} = 1 / [1 + C(R_o/R)^{2n}]$$

To apply frequency domain filtering to an image, use the *LoadData* command to open the *Earth Mantle Convection* image. This is a 248 by 248 two-dimensional array.

```
IDL> convec = LoadData(11)
```

Type these commands to open a window, load the Standard Gamma II color table, and display the original image in the upper left-hand corner:

```
IDL> Window, 0, XSize=248*2, YSize=248*2
IDL> LoadCT, 5
IDL> TV, convec, 0, 248
```

The first step in frequency domain filtering is to transform the image from the spacial domain into the frequency domain with the FFT function, like this:

```
IDL> freqDomainImage = FFT(convec, -1)
```

In general, low frequency terms represent the general shape of the image and high frequency terms add fine detail to the image. Viewing the frequency domain image is not usually instructive, but it is sometimes useful to observe the *power spectrum* of the frequency domain image.

The power spectrum is a plot of the magnitude of the various components of the frequency domain image. Different frequencies are represented at different distances from the origin (usually represented as the center of the image) and different directions from the origin represent different orientations of features in the original image. The power at each location shows how much of that frequency and orientation is present in the image. The power spectrum is particularly useful for isolating periodic structure or noise in the image. The power spectrum magnitude is usually represented on a log scale, since power can vary dramatically from one frequency to the next.

To calculate and display the power spectrum of this convection image next to the original image, type:

```
IDL> power = Shift(Alog(Abs(freqDomainImage)), 124, 124)
IDL> TV, power, 248, 248
```

The symmetry in the power spectrum indicates that this image contains a great deal of periodic structure at increasing frequencies. (Your output should look similar to the illustration in Figure 43. The purpose of this exercise is to filter out the higher frequencies in the image.

The next step is to apply a frequency filter to the transformed image. A Butterworth low-pass filter is used to filter out the high frequency components of the image. These

high frequencies give detail to the image, so the end result will be to perform an image smoothing operation. Construct the low-pass frequency filter by typing this:

```
IDL> filter = 1.0 / (1.0D + Dist(248)/15.0)^2
```

Notice that the cutoff frequency width is 15 pixels. This will be sufficient to eliminate about half the higher frequencies you see in the power spectrum in Figure 43.

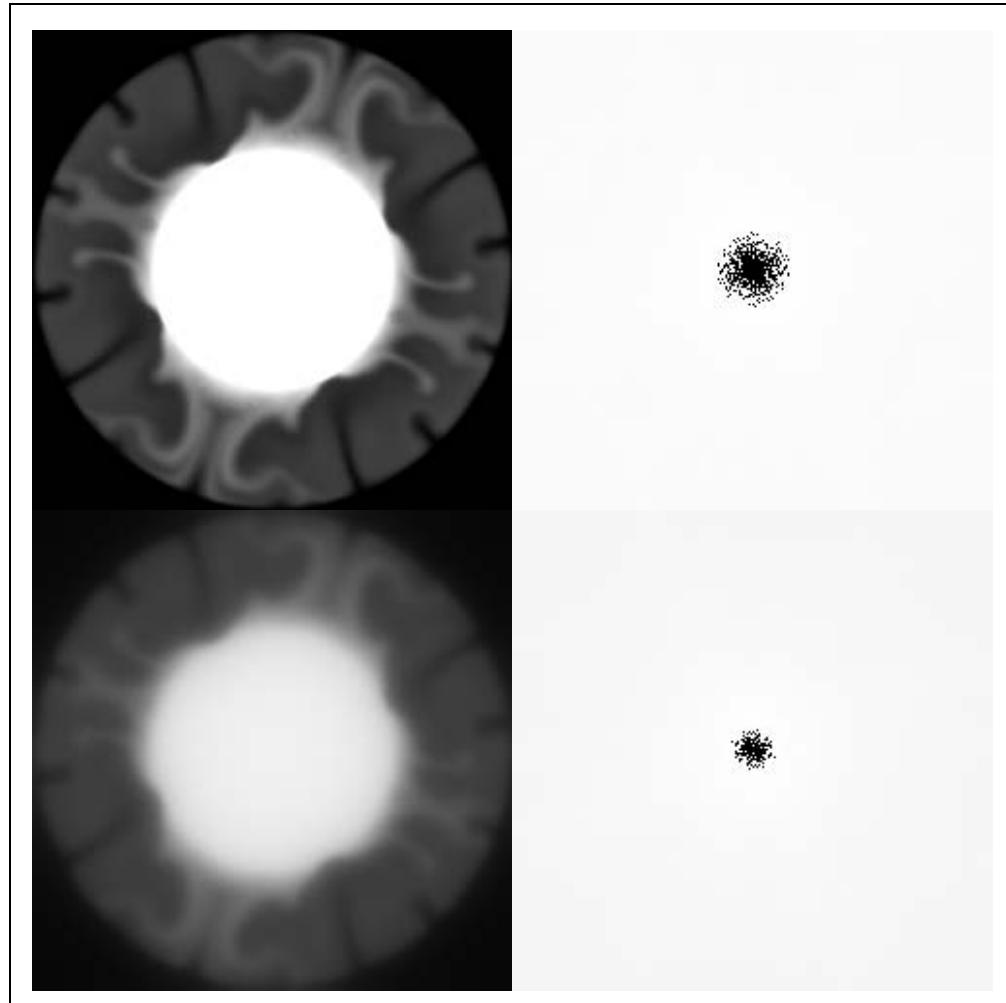


Figure 43: An illustration of frequency domain filtering. The unfiltered image next to its power spectrum in the top half of the figure. The filtered image next to its power spectrum in the bottom half of the figure. About half the high-frequency components have been removed from the filtered image.

To apply the frequency filter, transform the image from the frequency domain back into the spatial domain, and finally display the filtered image, type:

```
IDL> filtered = FFT(freqDomainImage * filter, 1)
IDL> TV, filtered, 0, 0
```

To prove you did in fact filter out the higher frequency components of the image, display the power spectrum of the filtered image next to the filtered image. Type:

```
IDL> filteredFreqImg = FFT(filtered, -1)
IDL> power = Shift(Alog(Abs(filteredFreqImg)), 124, 124)
IDL> TV, power, 248, 0
```

Your output will look similar to the illustration in Figure 43.

Chapter 4

♦ Discovering the Possibilities ♦♦♦



Graphical Display Techniques

Chapter Overview

After you have learned how to display a line, surface, and contour plot, you are on your way to displaying data in imaginative and innovative ways. This chapter describes a number of specific visualization techniques that will enhance your data displays. No attempt is made to describe every possible technique in IDL. Rather, this chapter introduces a few of the more common ones. The purpose of this chapter is to give you tools and ideas for creating your own unique data displays.

Specifically, you will learn:

- How IDL works with colors
- How to ask for color in a device independent way
- How to create and save color tables in IDL
- How to modify axis annotation to your specifications
- How to set up a 3D coordinate system in IDL direct graphics
- How to combine graphical displays
- How to work with bad or missing data in IDL
- How to animate graphical displays
- How to grid XYZ data for graphical display
- How to provide cursor interaction with your graphical display
- How to erase annotation from your graphical display
- How to draw a “rubberband” box on your graphical display
- How to use the Z graphics buffer for graphical display tricks

Working with Colors in IDL

A color in IDL is composed of three specific values. We call these values a color triple and write the color as $(red, green, blue)$, where the *red*, *green*, and *blue* values represent the amount of red, green, and blue light that should be assigned that color on the display. Each value ranges between 0 and 255. Thus, a color may be made up of 256 shades or gradients of red, 256 shades of green, and 256 shades of blue. This means that there are 256 times 256 times 256, or over 16.7 million colors that can be

represented on the display by IDL. To give an example, a yellow color is made up of bright red and bright green, but no blue color. The color triple that represents a yellow color is written as (255, 255, 0).

It used to be—with few exceptions—that color triples were accessed in IDL by using a number, called the *index*, to look up values in a table. These days, with more and more 24-bit graphics cards available, we are often expressing the color triple directly. If we do use an index, this look-up table is called a color translation table or—more often—just a *color table*. A color table consists of three columns of numbers. One column represents red values, one column represents green values, and one column represents blue values. Typically, these columns of numbers are represented as vectors. What you do when you load a color table in IDL is select exactly which numbers are placed into these columns or vectors. You see an illustration of this concept in Figure 44.

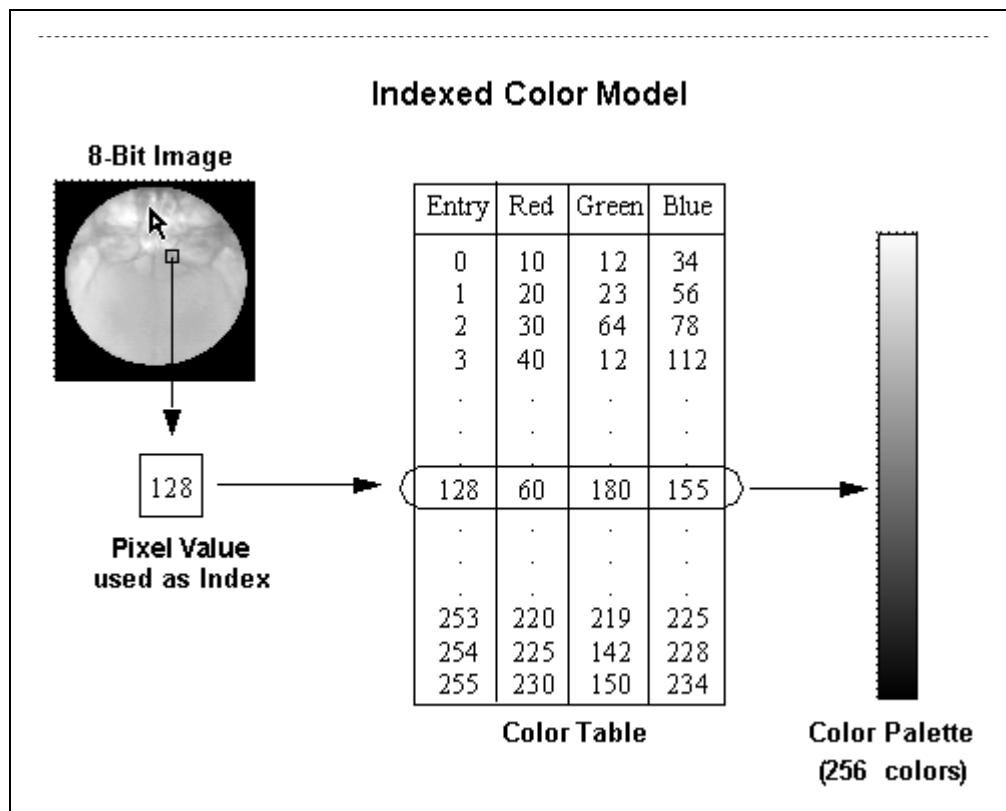


Figure 44: The Indexed Color Model for 8-bit pixel values. The pixel value is used as an index into the color table. The values found in the red, green, and blue columns of the color table determine the specific color triple associated with or indexed to that pixel value.

Using the Indexed versus the RGB Color Model

In addition to knowing that a color is represented as a color triple and that a color table is often used to determine what that color triple should be, you must also be aware of two color models that are used in IDL. The Indexed Color Model is used on 8-bit displays and the RGB Color Model is used on 24-bit displays. (IDL also uses a modified form of the RGB Color Model on PC's and Macintosh computers that support 16-bit color.)

Both models can use a color translation table to determine the specific color used on the display. (RGB Color Models use a color translation table if color decomposition is

turned off. Otherwise, they specify colors directly as a color triple.) But the Indexed Color Model also ties or links the indexed color to a specific location in the color table, whereas the RGB Color Model specifies the color directly. Colors that are linked to a particular color table location are called *dynamic* color displays. While colors that are displayed directly are often called *static* color displays. For the most part (there are exceptions), 8-bit displays are dynamic displays and 24-bit displays are static displays.

The most important difference between a dynamic and static color display is that if you have a dynamic display and you change the numbers loaded at a particular location in the color table, pixels that are indexed to that location change colors immediately. Whereas with a static display, pixel colors are specified directly and more or less permanently. They are unaffected by subsequent changes in the color table values. (This is not always true. See the discussion below concerning the DirectColor visual class.)

While this may seem strange, it actually has great value. What it means is that systems that use the RGB Color Model can display all 16.7 million colors simultaneously, whereas systems that use the Indexed Color Model can only display 256 simultaneous colors out of the palette of 16.7 million colors.

You can tell what type of color model you are using by using new keywords to the *Device* command that were first introduced in IDL 5.1. These keywords are *Get_Visual_Depth* and *Get_Visual_Name*. These are both output keywords which return a value to a named IDL variable, like this:

```
IDL> Device, Get_Visual_Name=thisName, $  
      Get_Visual_Depth=thisDepth  
IDL> Print, thisName, thisDepth  
      TrueColor          24
```

Visual names will normally be either *PseudoColor*, *DirectColor*, or *TrueColor*. The visual depth will normally be either 8, 16, or 24, which refers to the number of bits used to determine a specific color in this visual class.

An 8-bit PseudoColor visual class will indicate that you are using an Indexed Color Model and using a dynamic color display. A 24-bit TrueColor or DirectColor visual class will indicate that you are using the RGB Color Model. DirectColor visuals may sometimes be dynamic color displays, but this is a function of the Window manager and can occasionally be configured by the user. Usually, DirectColor visuals use a static color display. TrueColor visuals *always* use a static color display. (It wasn't until IDL 5.1 that this behavior was made consistent across all platforms running IDL.)

Static versus Dynamic Color Visuals

A PseudoColor visual class is a *dynamic* color visual. What this means is that if you change the color specified in the color look-up table, any pixel on the display that uses that color index changes color immediately. In general, loading a new color table will immediately change the colors of any graphic on the display.

A TrueColor visual class is a *static* color visual. This means that changing a specific color in the color table will have absolutely no effect on any graphic or color that is already on the display, since those colors were expressed directly as RGB triples.

DirectColor visual classes are more difficult to discuss. DirectColor visual classes are only available on UNIX machines. Each machine manufacturer seems to have slightly different ideas about what a DirectColor visual class means. But in theory the DirectColor visual class should give you the best of both worlds: a 24-bit color system that behaves as if it were a dynamic color visual. In practice, I've seldom seen it work this well. The most common problem is that DirectColor visuals often give you

private color maps, which require you to make the graphics window the current window to load the correct colors in the window. When this is done, other windows can disappear. This is known as the “color flashing problem” and is a result of the way the X Window manager handles color tables. Recent advances in both hardware and software has eliminated many of these problems, but they are still encountered frequently. As a rule, I like to use either 8-bit PseudoColor or 24-bit TrueColor visuals, since I can count on these working properly across many platforms.

The visual class used in an IDL session is normally assigned by default when IDL starts up, either from information in the *.XDefaults* file if IDL is running on a UNIX machine or from IDL’s normal rules for assigning the visual class. (The rules require that IDL inquire of the hardware what visual classes are supported and assigns the “highest” visual class and depth available.) But this default assignment can be overruled by specifying the visual class and depth from within IDL. (PC and Macintosh versions of IDL make their assignment from the graphics card installed on the machine and its current configuration. This cannot be changed from within IDL.) This assignment must be made *before* any graphics windows have been opened and will apply for the rest of the IDL session.

Here are typical visual class assignment statements on a UNIX machine:

```
IDL> Device, Pseudo_Color=8  
IDL> Device, True_Color=24
```

Specifying Colors on an 8-Bit Display

If you are using the Indexed Color Model, you specify a particular color as an index into the color table. IDL looks that color index up in the table and uses the values found in the red, green, and blue columns of the color table as the color triple that specifies a particular color. For example, suppose you load the color triple (255,255,0), which specifies a yellow color, into the color table at entry 180. You can do this with the *TVLCT* command, like this:

```
IDL> TVLCT, 255, 255, 0, 180
```

If you want to draw a plot in the yellow color, you specify the color index with the *Color* keyword, like this:

```
IDL> data = LoadData(1)  
IDL> Plot, data, Color=180
```

Similarly, any image pixel that has a value (i.e., index) of 180 will be displayed in the same yellow color.

If you are using the Indexed Color Model, you can easily change the color of the plot by simply loading a new color triple into entry 180 of the color table. For example, you can load a green color, like this:

```
IDL> TVLCT, 0, 255, 0, 180
```

You see the plot color change immediately because the color is indexed to the color table.

Specifying Decomposed Colors on a 24-Bit Display

If you are on a 24-bit display, however, the situation is slightly more complicated. By default, IDL uses “decomposed” color when it uses the RGB Color Model. That is to say, IDL does not treat the color index as a single index into the color table, but it tries to “decompose” the index into three separate indices into the color table. It does this by assuming that the index is a 24-bit long integer. IDL uses the lowest eight bits of the number as a red index, the middle eight bits as the green index, and the highest eight bits as the blue index. For example, on a 24-bit display with color decomposition

on, IDL tries to decompose the number 180 in the command above into red, green, and blue indices. Since the number 180 sets only bits in the red portion of a decomposed number, this plot will be displayed with a red color, no matter what color is actually loaded at that index in the color table. You see this idea of a decomposed index illustrated in Figure 45.

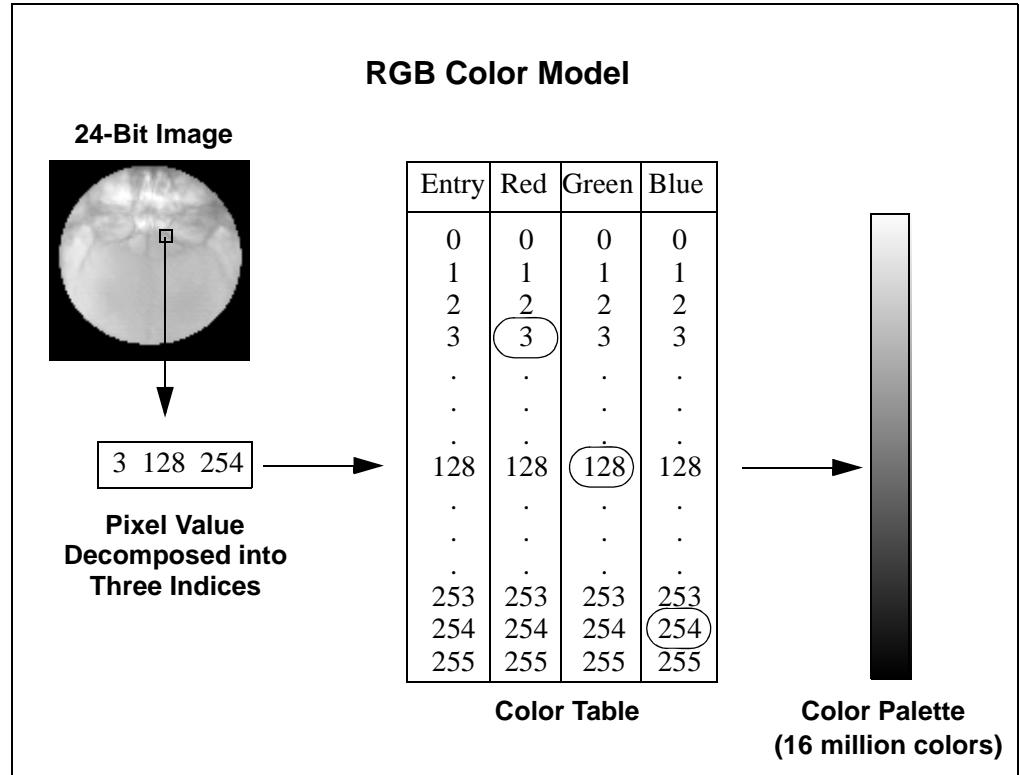


Figure 45: The RGB Color Model uses a 24-bit pixel value to independently specify the RGB components of a color. All 16.7 million colors are available simultaneously in the color palette if the color vectors contain values from 0 to 255.

When a gray-scale color table is loaded (as illustrated in Figure 45) all 16.7 million colors are immediately accessible to IDL. So, for example, if I wanted to draw a plot in the color yellow on this 24-bit system, I would want to pick a 24-bit integer number that had the eight lowest bits set (full red), the eight middle bits set (full green), and none of the highest bits set (no blue). As it happens, this number for the color yellow is represented as the long integer 65535. To draw the plot above on a 24-bit color display, you would type:

```
IDL> Plot, data, Color=65535L
```

Since most of us are not fluent manipulating the bits of a 24-bit number, the number is sometimes expressed in hexadecimal notation, where two digits (0-F) are enough to set eight bits at a time (i.e., 256 or 2^8 possible values can be set by two hexadecimal digits). For example, using hexadecimal notation, the way to express full red and green, but no blue is like this:

```
IDL> Plot, data, Color='00FFFF'xL
```

To draw a yellow (255, 255, 0) plot on a charcoal (70, 70, 70) background, with a green (0, 255, 0) title, using hexadecimal notation, you type this:

```
IDL> Plot, data, Color='00FFFF'xL, Background='464646'xL
IDL> XYOutS, 0.5, 0.95, Align=0.5, /Normal, 'Plot Title', $
```

```
Color='00FF00'xL
```

Since you may not be fluent in either 24-bit notation or in hexadecimal notation, you might want to use the program *Color24* to obtain a 24-bit integer value. This program came with the other programs you downloaded to use with this book and can convert any RGB triple (written as an IDL vector of three elements) into the equivalent 24-bit integer value. For example, on a 24-bit system, if you want to draw the plot in a yellow color using the *Color24* program, the command looks like this:

```
IDL> Plot, data, Color=Color24([255,255,0])
```

If you want to write code that will work on either an 8-bit or a 24-bit display, your code might look like this:

```
Device, Get_Visual_Depth=thisDepth
IF thisDepth GT 8 THEN BEGIN
    Plot, data, Color=Color24([255,255,0])
ENDIF ELSE BEGIN
    TVLCT, 255, 255, 0, 100
    Plot, data, Color=100
ENDELSE
```

Specifying Undecomposed Colors on a 24-Bit Display

You do not have to use decomposed color indexing with the RGB Color Model. For example, you might want to load color tables like you do on an 8-bit display and use the same code on both the 8-bit and 24-bit display. This is possible if you turn color decomposition off. This is done with the *Device* command, like this:

```
IDL> Device, Decomposed=0
```

In this case, IDL treats the pixel value or color index as if it were an 8-bit color index. That is to say, the index is used to access the same entry in the red, green, and blue color table vectors. Thus, on a 24-bit display, using undecomposed color, you can draw a yellow plot by typing these commands:

```
IDL> TVLCT, 255, 255, 0, 180
IDL> Plot, data, 180
```

But—and this is extremely important—the plot color is expressed directly. It is not indexed to the entry location in the color table. If you change the color table values at entry 10, the plot colors will be completely *unaffected*.

```
IDL> TVLCT, 0, 255, 0, 180
```

You will have to re-issue the *Plot* command (or, in general, re-display the graphic) to see the new colors take effect.

```
IDL> Plot, data, Color=180
```



On Windows machines prior to IDL 5.1, IDL *always* displayed 8-bit images on a 24-bit display as if it were using undecomposed color, no matter how color decomposition was configured. That is to say, the 8-bit pixel value was used to index the same entry in all three of the color table vectors. This behavior (a long-standing bug in the PC version) changed in IDL 5.1 to make it consistent with the behavior on other platforms. Among other things, this change made it just a little harder to write 8-bit image display programs that work identically in both 8-bit and 24-bit environments.

Determining if Color Decomposition is On or Off

As of IDL 5.1.1 there was no way to tell for sure whether color decomposition on a 24-bit display is turned on or off. This meant that if you wanted to have color decomposition one way or the other (e.g., you might be displaying either an 8-bit image, in which case you want color decomposition to be turned off, or a 24-bit

image, in which case you want color decomposition turned on) you must set it before you issue the graphics display command:

```
Device, Decomposed=0
TV, image8bit
Device, Decomposed=1
TV, image24bit, True=1
```

A new *Get_Decomposed* keyword was introduced for the *Device* command in IDL 5.2 which can tell you the current “decomposed” state.

```
IDL> Device, Get_Decomposed=usingDecomposed
IDL> Print, usingDecomposed
```



Note that if you are running IDL on a PC or Macintosh platform with a 24-bit display, and you want to display a 24-bit image in the correct image colors, you must either have a gray-scale color table loaded or you must set the *Decomposed* keyword equal to 1. If you have *Decomposed* set equal to 0, then the 24-bit image values are routed through the color table vectors. This *never* results in what you want, unless a gray-scale color table is loaded. So a good rule of thumb is to always set color decomposition on before you display a 24-bit image. Note that the *TVImage* program you downloaded with this book automatically sets the correct *Decomposed* value depending upon whether the image is 8-bit (*Decomposed=0*) or 24-bit (*Decomposed=1*).

Obtaining Device Independent Colors

As you can see, working with colors in IDL can be quite complicated. And it is made even more so by the fact that the way colors work in IDL has been changing with almost every release of IDL since 5.0. There are a *lot* of things to keep track of!

To help you make sense of this, and to help you write code that is device independent, a number of programs have been included with this book that make it easier to write programs that can run in any IDL environment. In fact, this is one of the goal’s of this book. You have already been introduced to one of these programs: *TVImage*, a program that can display 8-bit or 24-bit images in a device independent fashion. Another of these programs is *GetColor*, which can be used to obtain a color reference in a device independent way.

Specifically, *GetColor* allows you to ask for a particular color by name. You can see the names of the 16 colors *GetColor* “knows” about by typing these commands:

```
IDL> names = GetColor(/Names)
IDL> Print, names
Black Magenta Cyan Yellow Green Red Blue Navy Pink Aqua
Orchid Sky Beige Charcoal Gray White
```

And you can see the colors themselves, by using a companion program, named *PickColor*, to display them:

```
IDL> color = PickColor(/Names)
```

GetColor works like this. If it is called with a single parameter that is the “name” of a color it recognizes, it returns either a three-element vector that is that color’s triple or a 24-bit value that can be decomposed into that color, depending upon the current decomposed state of the device at the time the program is called. If you have color decomposition turned off (or if you are on an 8-bit display device) then you can load the color triple at a color table index and use it appropriately, like this:

```
IDL> Device, Decomposed=0
IDL> yellow = GetColor('yellow')
IDL> TVLCT, yellow, 180
```

```
IDL> Plot, LoadData(17), Color=180
```

If you are on a 24-bit device, with color decomposition turned on, you can use the return value directly, like this:

```
IDL> Device, Decomposed=1
IDL> yellow = GetColor('yellow')
IDL> Plot, LoadData(17), Color=yellow
```

In fact, you could combine the last two commands into one, like this:

```
IDL> Plot, LoadData(17), Color=GetColor('yellow')
```

That's easy enough and you have the advantage of seeing in your code exactly which color you are using, but you still have to be sure of your current decomposed state and you can't really do that on an 8-bit device or with color decomposition turned off, since you must first load the color into the color table. Isn't there a way to make *GetColor* to do this automatically?

Of course. But what you have to do is tell *GetColor* where it should load the color in the color table. So, you must call *GetColor* with two parameters: the name of the color and a location in the color table where the color should be loaded. I like to load drawing colors at the top of the color table, but not at the very top, since many programs assume that color will be white. I like to load a drawing color in the next to last element in the color table vector. I would call *GetColor* like this:

```
IDL> yellow = GetColor('yellow', !D.Table_Size-2)
```

The return value is either the index number where the color was loaded (i.e., *!D.Table_Size-2*), if color decomposition is off or you are on an 8-bit display device, or a 24-bit value that can be decomposed into the appropriate color if you are on a 24-bit device and color decomposition is turned on. This means that these two commands will work appropriately no matter what kind of device we are on or what the current color decomposition state:

```
IDL> yellow = GetColor('yellow', !D.Table_Size-2)
IDL> Plot, LoadData(17), Color=yellow
```

Or, you can even put these two commands together into a single command and have it work appropriately in all IDL environments. For example:

```
IDL> Plot, LoadData(17), $
      Background=GetColor('charcoal', !D.Table_Size-2), $
      Color=GetColor('yellow', !D.Table_Size-3)
```

Having programs that work correctly everywhere is, of course, a desirable goal in *any* kind of programming, not just IDL programming, but it is not always as easily achieved as this.

Loading Color Tables on a 24-Bit Display

You are aware, now, that when you use the Indexed Color Model on an 8-bit display that the pixel colors are indexed directly to the color table. In other words, if you change the values in the color table by loading a color table in IDL, the colors associated with those indices change too. If you want to display several images at the same time using different color tables for each, then you must divide the available indices of the color table into different regions with different colors loaded in each region. (See “Scaling Images into Different Portions of the Color Table” on page 63.) As a practical matter, you can probably only have four or five images at most with different color tables before you run out of index numbers.

One of the huge advantages of a 24-bit display is that you can have literally dozens of images on the display at once, each displayed with a different color table. (Remember

that loading color tables on a 24-bit display only really makes sense if color decomposition is turned *off*. And remember that it is turned *on* by default when IDL starts up.)

But what if you now loaded a different color table? Would you want the colors in those dozens of images to change? Probably not, since each image uses its own set of colors, which are specified *directly*.

Thus, if you have an image displayed on a 24-bit monitor, and you change the color table with a color table changing tool like *XColors* or *XLoadCT*, the new colors will not take effect until you *re-display* the image, and translate the image pixels once again through the color table into specific direct colors. To see how you might write code to change color tables on a 24-bit display and have your graphics re-displayed automatically, see “Automatic Updating of Graphic Displays When Color Tables are Loaded” on page 66.

Obtaining a Copy of the Color Table

There are two ways to obtain a copy of the red, green, and blue values in the current color table. One way is to declare the common block *Colors*, either at the main IDL level if you want the color table available there, or in any IDL procedure or function. The call looks like this:

```
COMMON Colors, r_orig, g_orig, b_orig, r_cur, g_cur, b_cur
```



Note that a color table must have been loaded in the IDL session for the variables in the common block to be defined.

The convention is to get the colors of the current color table from the first three variables. Then if you modify those colors, you place the modified color vectors into the last three variables. To load the color table, you use the *TVLCT* command. If you wanted to reverse the colors in the color table, your code might look like this:

```
IDL> COMMON Colors, r, g, b, rr, gg, bb
IDL> rr = Reverse(r)
IDL> gg = Reverse(g)
IDL> bb = Reverse(b)
IDL> TVLCT, rr, gg, bb
```

Another way to get the values of the color table is to use the *TVLCT* command with the *Get* keyword, like this:

```
IDL> TVLCT, red, green, blue, /Get
```

In this instance, the variables *red*, *green*, and *blue* are output variables and are filled with the appropriate values from the color table. Note that these variables contain as many elements as the number of colors you are using in the IDL session. You can determine how many colors you are using in the IDL session by first opening an IDL graphics window and then typing:

```
IDL> Print, !D.Table_Size
```

Modifying and Creating Color Tables

There are two principal commands to manipulate the colors in the color table: *XLoadCT* and *XPalette*. Between the two, you should be able to make the colors in a color table exactly the way you want them. *XLoadCT* allows you to stretch the colors in various ways. (Go into *Function* mode and click the *Add Control Point* button several times. Move a control point with the mouse to see how you can affect the color table.) It also allows you to change the Gamma correction interactively. (A Gamma value of one is a linear ramp function from one value to the next. Gamma values of

less than one or greater than one result in exponential ramp functions of different shapes and steepness.)

The *XPalette* command allows you to modify and create your own color tables by setting end point colors with sliders, and then interpolating the intervening values. Individual colors may also be modified in this program. Note that you may specify colors in *XPalette* in color systems other than the RGB color system. In the end, however, no matter what color system you use to specify the colors, it is a red, green, and blue color vector that is loaded into the color tables.



Note that if you run *XPalette* on a 24-bit display you should be sure that you have turned color decomposition off, since this program is an old 8-bit program that has not been updated to work automatically on 24-bit devices.

It is quite easy to make your own color tables as well. Here is a simple little program (with no error checking!) named *Make_CT* that can create a color table of an arbitrary number of colors between two color end points. Open a text editor and type:

```
FUNCTION MAKE_CT, begColor, endColor, ncolors
scaleFactor = FindGen(ncolors) / (ncolors - 1)
colors = BytArr(ncolors, 3)
FOR j=0,2 DO colors[*,j] = begColor[j] + (endColor[j] $
- begColor[j]) * scaleFactor
RETURN, colors
END
```

Compile this program by typing this:

```
IDL> .Compile make_ct
```

Open the *World Elevation Data* image and display it in a window, like this:

```
IDL> image = LoadData(7)
IDL> Window, XSize=360, YSize=360
IDL> LoadCT, 0
IDL> TVScl, image
```

Suppose you want a color table that goes from yellow (255, 255, 0) to blue (0, 0, 255). You can create it with the *Make_CT* program and load it like this:

```
IDL> yellow = [255, 255, 0]
IDL> blue = [0, 0, 255]
IDL> TVLCT, Make_CT(yellow, blue, !D.Table_Size)
IDL> Device, Decomposed=0
IDL> TVScl, image
```

Suppose you want to display this image in 150 colors that go from yellow to green to blue. You might do this:

```
IDL> scaledImage = BytScl(image, Top=149)
IDL> green = [0, 255, 0]
IDL> TVLCT, Make_CT(yellow, green, 75)
IDL> TVLCT, Make_CT(green, blue, 75), 75
IDL> TV, scaledImage
```

Note that a lot of obfuscation of data can take place in the way you choose your colors. Be careful. You might want to have a look at the paper by Bernice E. Rogowitz and Lloyd A. Treinish entitled “How Not to Lie with Visualization” in *Computers In Physics*, 10(3):268, 1996. If you are really interested in color and the display of data, read *The Visual Display of Quantitative Information* and *Envisioning Information* by Edward Tufte. These books just might change the way you write IDL programs!

Saving Your Own Color Tables

Suppose you are happy with the color table you just created and you want to save it. You can get the color values with the *TVLCT* command, like this:

```
IDL> TVLCT, r, g, b, /Get
```

Look at how long these vectors are. Type:

```
IDL> Help, r, g, b
```

You see that they are as long as the number of colors that you are using in your IDL session. But the colors for the color table above are only in the first 150 values. Redefine the vectors with just that number of values, like this:

```
IDL> r = r(0:149)
IDL> g = g(0:149)
IDL> b = b(0:149)
```

You could save the vectors now if you wanted to, but most color table vectors are 256 elements in length. It would be easy to make these vectors that length. Type:

```
IDL> r = Congrid(r, 256, /Interp)
IDL> g = Congrid(g, 256, /Interp)
IDL> b = Congrid(b, 256, /Interp)
```

Now, you can write these vectors into a file if you like, but it is easier to use IDL's *Save* command to save them in an IDL save file. Type:

```
IDL> Save, File='mycolors.sav', r, g, b
```

Next, just to prove to yourself that you did save the vectors, load another color table and delete these three variables. Type:

```
IDL> LoadCT, 0
IDL> DelVar, r, g, b
IDL> Help, r, g, b
```

Now, when you are ready to use the vectors, you restore them with the *Restore* command. Notice that they come back in the same variable name in which you saved them. If you have a variable with the same name defined in the same IDL session (as you do here), the variable will be overwritten. This means you will probably want to give these variables well-thought out names. Type:

```
IDL> Restore, 'mycolors.sav'
IDL> Help, r, g, b
```

To use these variables you will have to resize them into the number of colors in your IDL session. The commands will look like this:

```
IDL> r = Congrid(r, !D.Table_Size)
IDL> g = Congrid(g, !D.Table_Size)
IDL> b = Congrid(b, !D.Table_Size)
IDL> TVLCT, r, g, b
```

Rather than type these commands every time you want to load one of your saved color tables, it would be easier to write a little IDL program that did it automatically. If you always save your RGB vectors with the variable names *r*, *g*, and *b*, you can write a program name *CT_Load* like this. (There is no error checking in this little program!) Open a text editor and type:

```
PRO CT_Load, filename
IF N_Params() EQ 0 THEN filename = 'mycolors.sav'
Restore, filename
r = Congrid(r, !D.Table_Size)
```

```
g = Congrid(g, !D.Table_Size)
b = Congrid(b, !D.Table_Size)
TVLCT, r, g, b
END
```

Save the file as *ct_load.pro* and compile it like this:

```
IDL> .Compile ct_load
```

Now, whenever you want to load this color table, all you have to do is type this:

```
IDL> CT_Load
```

Creating Your Own Axis Annotations

Sooner or later, the basic axis annotation that IDL applies to axes by default is not adequate for the display you have in mind. Fortunately, IDL provides a number of ways to augment the basic annotation properties of axes. This section describes of few of the techniques you can use to create more complex axis annotations.

Adjusting Axis Tick Intervals

Sometimes IDL's internal axis annotation algorithm will not divide the axis in a way that is best for your data. You can control the number of major tick intervals with the *[XYZ]Ticks* keyword. Load the *Time Series Data* that came with this book to have an example data set to work with. Type these commands:

```
IDL> curve = LoadData(1)
IDL> LoadCT, 5
IDL> Plot, curve
```

Notice that the X axis is divided into five major tick intervals. You can change this to ten major intervals by typing this:

```
IDL> Plot, curve, XTicks=10
```

Your output will look like the illustration in Figure 46.

Notice that the number of minor tick marks has also increased, resulting in a bit of a cluttered look to the axis. Since you are probably not interested in such a fine axis granularity, you might want to change the number of minor tick marks as well. The number of minor tick marks is set with the *[XYZ]Minor* keyword. For example, you may want just one minor tick mark between each major tick interval.

You might be tempted to set the *XMinor* keyword to 1 to get one minor tick mark between each major interval, but this would be incorrect. If you set the *XMinor* keyword to 1 all minor tick marks are *suppressed*. To get the number of minor tick marks you want, you actually set the *XMinor* keyword to a value that is *one more* than you want. (I didn't write the code!) To get just one minor tick mark, you type this:

```
IDL> Plot, curve, XTicks=10, XMinor=2
```

Formatting Axis Annotations

Another way you can affect axis annotation is to change the way axis labels are formatted. For example, the X axis labels are now expressed as integers. You might want to express them always as three-digit integers. You can do this by setting the *XTickFormat* keyword to the specific format you desire. For example, you can type this:

```
IDL> Plot, curve, XTickFormat='(I3.3)'
```

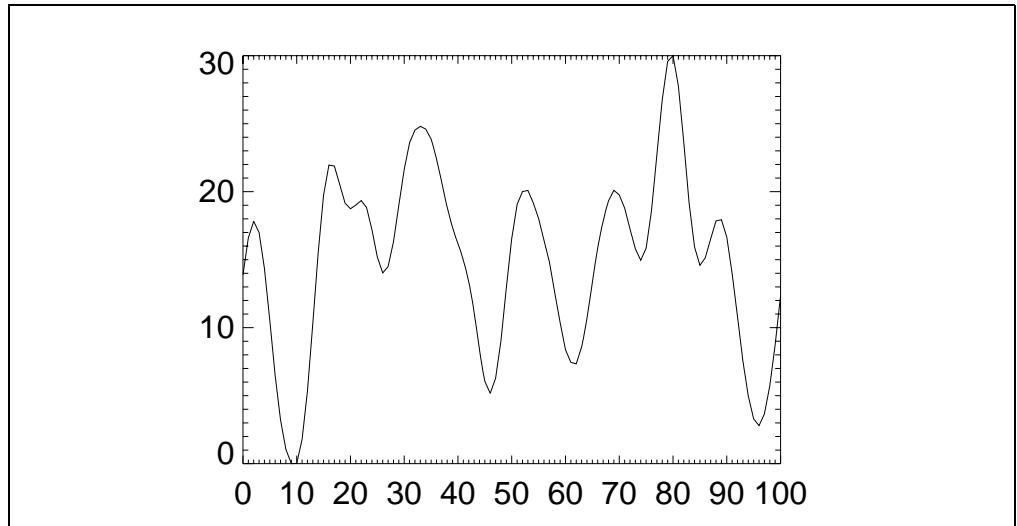


Figure 46: The number of major tick intervals is changed with the `XTicks` keyword.

To write the labels as floating point values with two digits to the right of the decimal place, type:

```
IDL> Plot, curve, XTickFormat='(F6.2)'
```

It is possible to use specific strings as tick labels, too. This is accomplished with the `TickName` keyword, which can have up to 30 string elements. For example, you can label the plot by the days of the week, like this:

```
IDL> labels = ['MON', 'TUE', 'WED', 'THU', 'FRI', 'SAT']
IDL> Plot, curve, XTickName=labels
```

Your output looks like the illustration in Figure 47.

The axis annotation can be suppressed by setting the axis tick format to (A1), like this:

```
IDL> Plot, curve, XTickFormat='(A1)'
```

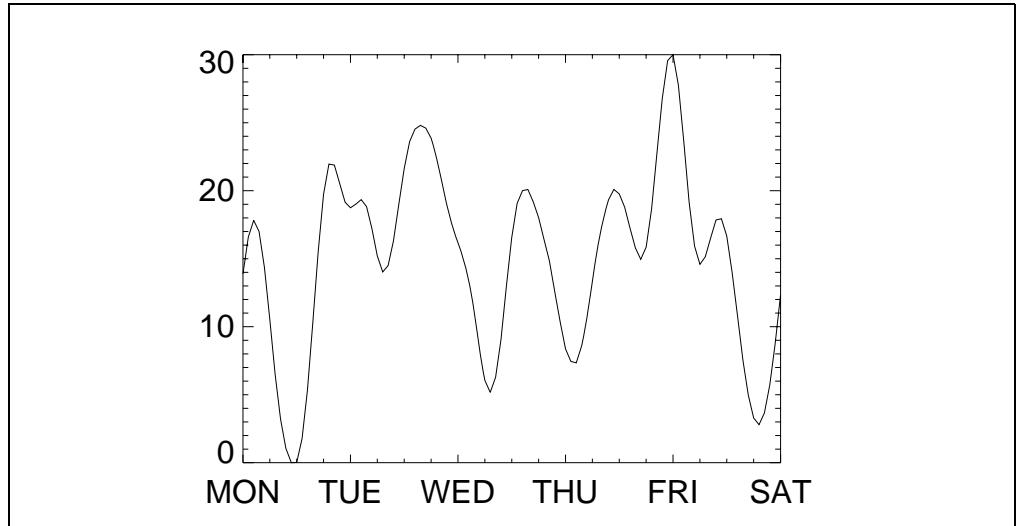


Figure 47: You can label your axes with strings by means of the [XYZ]TickNames keywords.

Writing a Tick Format Function

Another way to format tick labels is to write a function to format the tick labels exactly the way you would like them to be formatted. If the argument to the *[XYZ]TickFormat* keywords is a function name, then IDL will call that function when it applies the annotation to the label.

For example, suppose you want the X labels on this plot to be dates, written like this: *25 MAR 97*. You can write a function named *Date* to perform the formatting for you. The function must be defined with three and only three positional parameters. These will be the *axis number*, the *index number*, and the *label value*. IDL will call the function with these three positional parameters when it needs to format the axis. The return value of the function *must* be a string variable.

The *axis number* is 0, 1, or 2 to indicate whether this is the X, Y, or Z axis, respectively. The *index number* is the number of the particular axis label. This number is hardly ever used by the programmer inside the function. The *label value* is the normal value that would be used for the axis label. It is your job to use the label value to calculate or format a new value, which you return as the result of the function. It is this return value, then, that is used to label the axis for the particular axis index number.

It is easier to understand how to write this program with an example. Open a text editor and type this short *Date* program. (The program is among the files you downloaded to use with this book.)

```
FUNCTION DATE, axis, index, value
monthStr = ['Jan','Feb','Mar', 'Apr', 'May', 'Jun', 'Jul', '$
    'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
CalDat, LONG(value), month, day, year
year = StrMid(StrTrim(year,2), 2, 2)
RETURN, StrTrim(day, 2) + ' ' + monthStr(month-1) + ' ' $
    + year
END
```

Compile the *Date* program so that it can be used to format the X axis tick labels in the code below. Type:

```
IDL> .Compile date
```

Notice the *CalDat* command in this program. This program accepts a Julian number that represents a particular date and returns the number of the proper day, month, and year associated with the Julian number. This information is what you use to format the label properly. To see how this works, type:

```
IDL> Window, XSize=500, YSize=350
IDL> startDate = Julday(1, 1, 1991)
IDL> endDate = Julday(6, 23, 1995)
IDL> numTicks = 5
IDL> sizeCurve = N_Elements(curve)
IDL> steps = Findgen(sizeCurve) / (sizeCurve-1)
IDL> dates = startDate + (endDate+1 - startDate) * steps
IDL> !P.Charsize = 0.8
IDL> Plot, dates, curve, XTickFormat='Date', $
    XStyle=1, XTicks=numTicks, $
    Position=[0.15, 0.15, 0.85, 0.95]
```

Your output will look similar to the illustration in Figure 48. To learn more about labeling axes with dates, see the function *Label_Date* in the IDL library. The *Label_Date* function works much like the *Date* program you just wrote.

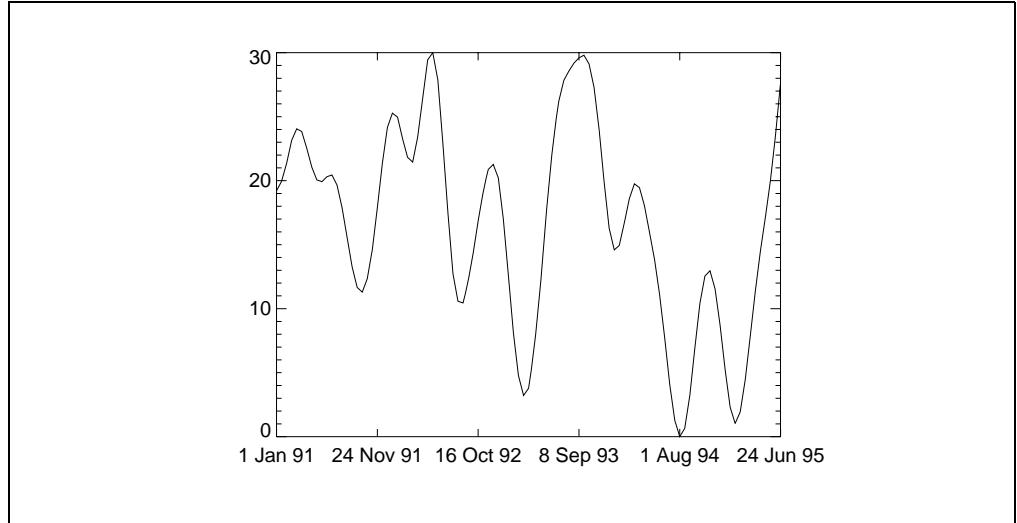


Figure 48: You can format tick labels by means of a user-written function.

As you can see, these labels are quite long and there is a danger that they will crowd together. You may want to display the dates in another way. For example, you might want to rotate them by 45 degrees with respect to the axis. Unfortunately, the tick formatting function you just wrote will not help in this case. You will have to resort to a more brute force method, placing the labels with the `XYSOut`s command.

You can, however, still use the *Date* program to format the strings for you. To make this work, you will have to draw the plot with the X axis labels suppressed and you will need a vector with the proper tick values. You can suppress the axis labeling by setting the tick format for the axis to *(A1)*. You can get the tick label values in vector form by using the *XTick_Get* keyword. For example, the plot will be drawn like this (the *Position* keyword is used to leave room for the axis labels):

```
IDL> Plot, dates, curve, XTickFormat='(A1)', XStyle=1, $  
      XTicks=numTicks, XTick_Get=tickValues, $  
      Position=[0.1, 0.2, 0.85, 0.95]
```

Then the labels will be attached with the *XYOutS* command. You can use the *!/[XY].Window* system variables to find the end-points of the X and Y axes in normalized coordinates. This information is critical for positioning the labels correctly. Your code will look like this:

```

IDL> ypos = Replicate(!Y.Window[0] - 0.04, numticks+1)
IDL> xpos = !X.Window[0] + (!X.Window[1]) - !X.Window[0]) * $
        FIndGen(numTicks+1)/numTicks
IDL> FOR j=0,numTicks DO XYOutS, xpos[j], ypos[j], $
        Date(0, j, tickValues[j]), Alignment=0.0, $
        Orientation=-45, /Normal

```

Your output will look like the illustration in Figure 49.

Handling Missing Data in IDL

Data, unfortunately, does not always come from your instrument in pristine condition. It is often necessary to manipulate the raw data to put it into a form you can use. In fact, many times the raw data set isn't even complete. Many things could have happened. The data collecting instrument shut down for a short period. An electrical fluctuation caused spurious data values. The operator mis-adjusted a control. What can you do with this kind of missing or bad data in IDL?

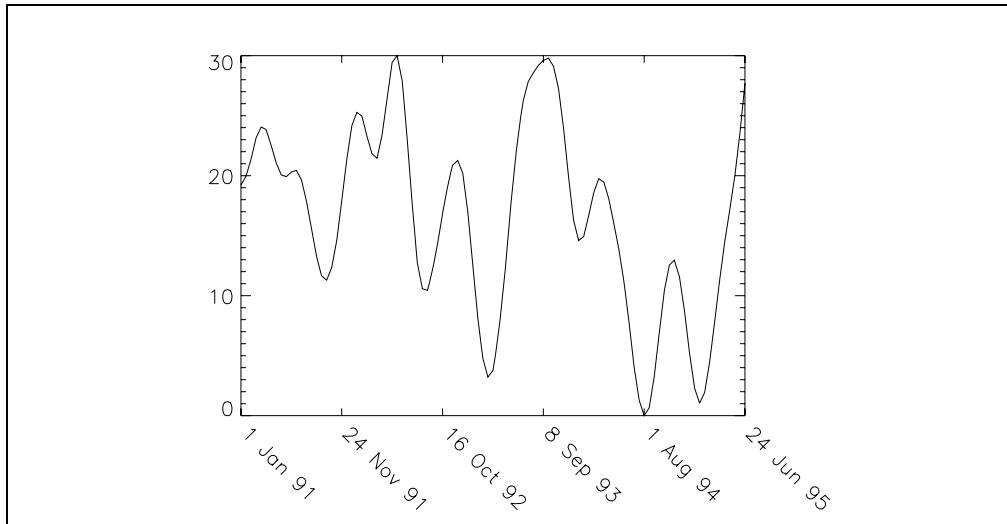


Figure 49: Rotated axis labels are created with the *XYOutS* command.

One way to handle this kind of data is to assign it the value *NaN*. The *NaN* value is a particular bit pattern, different on each machine architecture. The bit pattern for the machine IDL is running on is stored in the system variable *!Values* in the field *F_NaN*.

To see this, open the *Elevation Data* data set with the *LoadData* command, like this:

```
IDL> data = LoadData(2)
IDL> !P.CharSize = 1.0
```

This data set is a 41 by 41 floating point array. But suppose the data was not complete. Suppose while you were collecting the data the collecting instrument temporarily shut down while scanning three lines in the middle of the 2D data set. You want to assign the *NaN* value to these three scan lines. You can type this:

```
IDL> badData = data
IDL> badData[*, 30:32] = !Values.F_NaN
```

Now, when you display the surface, IDL does not connect those values that are represented by the *NaN* bit pattern. Type:

```
IDL> Surface, badData
```

Your output will look similar to the illustration in Figure 50.

There is another way to handle missing or bad data besides setting it to the *NaN* bit pattern. This is with the *Min_Value* and *Max_Value* keywords that are available for most IDL graphics output commands. By setting these keywords, any data value that is less than the minimum value or greater than the maximum value is ignored (not drawn) by the graphics output command. For example, in the elevation data set you used above, you can draw just those contours within a specific range. Here are some commands that show you how this works. Here contours lines with values less than or equal to 400 and greater than or equal to 1000 are not drawn in the plot on the right:

```
IDL> Window, XSize=500, YSize=375
IDL> !P.Multi = [0, 2, 1, 0, 1]
IDL> values = FIndGen(10)*150 + 100
IDL> label = Replicate(1,10)
IDL> Contour, data, Levels=values, /Follow, C_Labels=label
IDL> Contour, data, Levels=values, /Follow, C_Labels=label, $
      Min_Value=400, Max_Value=1000
IDL> !P.Multi = 0
```

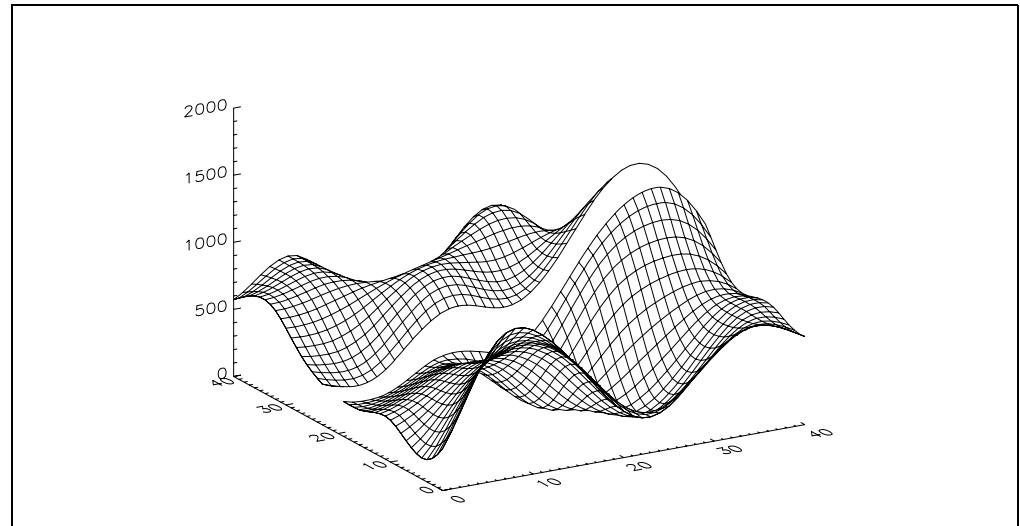


Figure 50: Missing data is represented in this surface plot by the NaN value.

Your output in the right-hand side of your graphics window will look similar to the illustration in Figure 51.

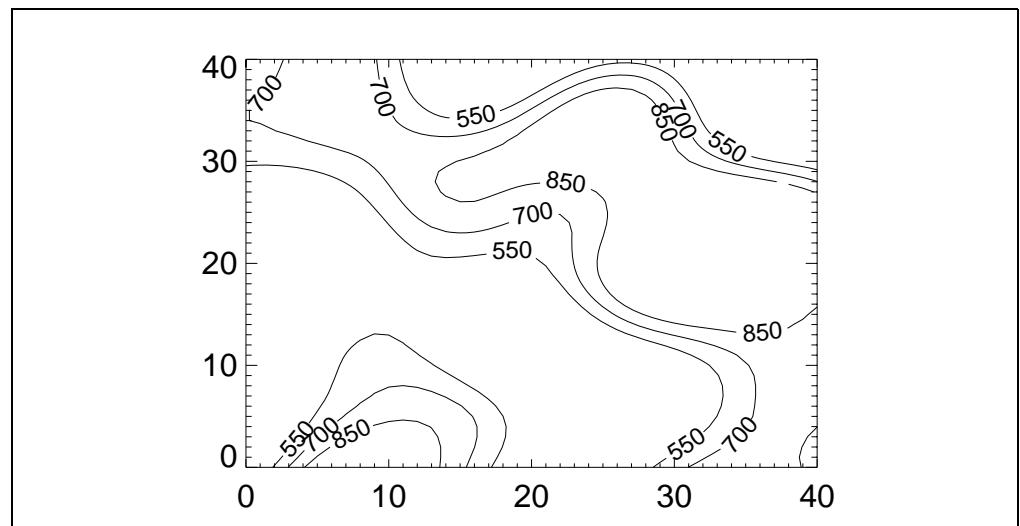


Figure 51: Contour lines can be suppressed by the Min_Value and Max_Value keywords to the Contour command.

Setting Up a 3D Coordinate System in IDL

IDL simulates a 3D coordinate system on the 2D display by multiplying each point in 3D space by a transformation matrix. This transformation matrix, if there is one, is stored in the `!P.T` system variable. If you want to draw graphics in a 3D space in IDL, you must first load the proper transformation matrix into the `!P.T` system variable, and then you must make sure your graphics output command is multiplied by the matrix before it appears on the display. This is quite easy to do in practice.

There are several ways to load the 3D transformation matrix into the `!P.T` system variable. If you want great control over how the matrix is set up, you can use the `T3D` command to set up the 3D coordinate system just exactly the way you want it. But unless you are doing something elaborate or out of the ordinary, you won't bother with the `T3D` command. Instead, you will load the 3D transformation matrix in one of two

ways: (1) use the *Surface* command with the *Save* keyword if you want to have axes in your 3D space, or (2) use the *Scale3* command if you just want a 3D space, but you are not concerned about axes.

Setting Up a 3D Scatter Plot

For example, suppose you have randomly-distributed 3D data and you want to display it as a scatter plot in 3D space. You can get some randomly-distributed 3D data to work with by typing:

```
IDL> seed = 3L
IDL> x = RandomU(seed, 41)
IDL> y = RandomU(seed, 41)
IDL> z = Exp(-4 * ((x - 0.5)^2 + (y - 0.5)^2))
```

To see that this is randomly-distributed data, type:

```
IDL> Window, XSize=500, YSize=350
IDL> Plot, x, y, PSym=4, SymSize=2.0
```

In this case, you probably want to have a set of axes to define your 3D space, so the *Surface* command with the *Save* keyword set will be a good choice for setting up the 3D transformation matrix. The *Save* keyword takes the 3D transformation matrix that is created for the *Surface* command and instead of throwing it away as it usually does, it saves it in the *!P.T* system variable. You can use the usual axis rotation keywords to get the 3D space the way you want it. The *NoData* keyword makes sure only the axes are displayed. The *[XYZ]Range* keywords are needed to set the 3D space up to reflect the correct range of the real data, not the dummy data being given to the *Surface* command. Type this:

```
IDL> Surface, Dist(10), /Save, /NoData, CharSize=1.5, $
      XRange=[0,1], YRange=[0,1], ZRange=[0,1]
```

This command sets up the normal three surface axes. You may want to include additional axes. For example, you may want to emphasize the XY plane. You could add an additional X and Y axis with the *Axis* command, like this:

```
IDL> Axis, YAxis=1, 1.0, 0.0, 0.0, /T3D, CharSize=1.5
IDL> Axis, XAxis=1, 0.0, 1.0, 0.0, /T3D, CharSize=1.5
```

To draw the points in this 3D space, you have to make sure each 3D point is first multiplied by the transformation matrix. This is accomplished by setting the *T3D* keyword on the graphics output command. In this case, you will be using the *PlotS* command, like this:

```
IDL> PlotS, x, y, z, PSym=4, SymSize=2.0, /T3D
```

To give this plot more of a 3D appearance, you might want to connect each point by a line to the XY plane. In fact, you might want to color the line according to the Z value, to give the user even more information. You might type this:

```
IDL> zcolors = BytScl(z, Top=99) + 1B
IDL> LoadCT, 22, NColors=100, Bottom=1
IDL> FOR j=0,40 DO PlotS, [x[j], x[j]], [y[j], y[j]], $
      [z[j], 0], Color=zcolors[j], /T3D
```

Your output will look similar to the illustration in Figure 52.

A graphical display with colors may be considered incomplete without a color bar to indicate what the colors mean. You can add a colorbar to this display with the *Colorbar* program you received with this book. Type:

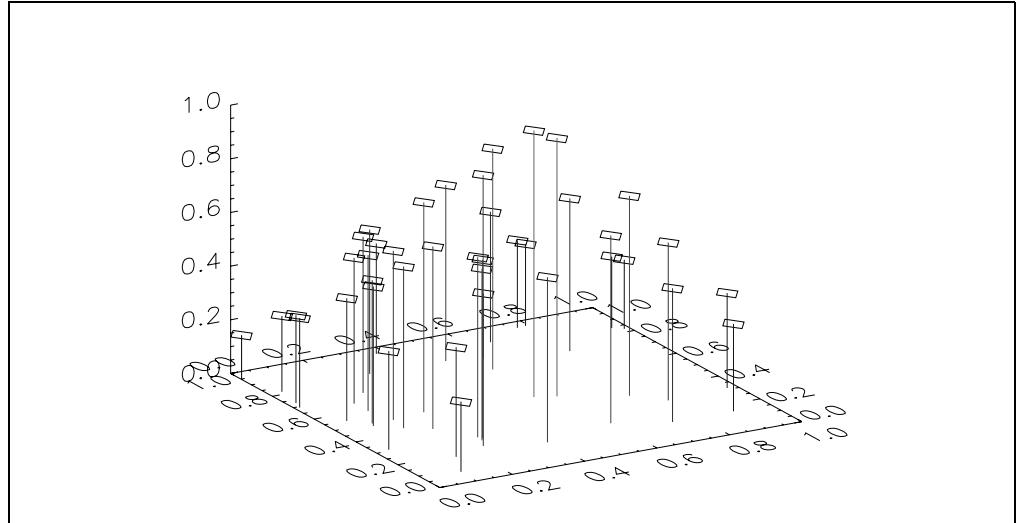


Figure 52: A scatter plot in the 3D space set up with the *Surface* command.

```
IDL> Colorbar, Position=[0.25, 0.9, 0.85, 0.95], $  
      Range=[Min(z), Max(z)], NColors=100, Bottom=1, $  
      Color=255, Title='Z Values'
```

Positioning the 3D Axes Through the Origin of a Plot

Here is another example of setting up a 3D coordinate system. In this example you have some data that spans the origin. You want the axes that define the data to be drawn through the origin. Since you don't want to draw the axes that belong to the data, this is an example where you might want to use the *Scale3* command to set up the 3D space. The *Scale3* command uses the same rotation matrix that is used by the *Surface* command, but it doesn't draw the axes and it loads the transformation matrix into the *!P.T* system variable by default.

To create some data for this example, load the 41 by 41 *Elevation Data* data set with the *LoadData* command, like this:

```
IDL> data = LoadData(2)  
IDL> data = data - (Max(data) / 2.0)  
IDL> x = FIndGen(41) - 20.0  
IDL> y = FIndGen(41)*2.0 - 41.0
```

Set up the 3D space with the *Scale3* command. To make the output as wide as possible, turn off the normal margins of the plot. Before you do so, save the current system variable set-up so it is easily restored later. Type:

```
IDL> Window, XSize=500, YSize=350  
IDL> Save, /System_Variables, File='system.sav'  
IDL> !X.Margin = 0 & !Y.Margin = 0 & !Z.Margin = 0  
IDL> Scale3, XRange=[Min(x), Max(x)], Ax=45, Az=45, $  
      YRange=[Min(y), Max(y)], ZRange=[0, Max(data)]
```

Now draw the surface plot of the data. Be sure to turn off the axis display on the *Surface* command with the *[XYZ]Style* keywords and force the *Surface* command to use the 3D transformation matrix you just created rather than calculating its own. Type:

```
IDL> Surface, data, x, y, /T3D, XStyle=4, YStyle=4, ZStyle=4
```

Now draw the axes through the origin. Type:

```
IDL> TVLCT, 255, 255, 0, 1  
IDL> Axis, 0, 0, 0, Color=1, /T3D, CharSize=2, XAxis=1  
IDL> Axis, 0, 0, 0, Color=1, /T3D, CharSize=2, YAxis=1  
IDL> Axis, 0, 0, 0, Color=1, /T3D, CharSize=2, ZAxis=1
```

Your output should look like the illustration in Figure 53.

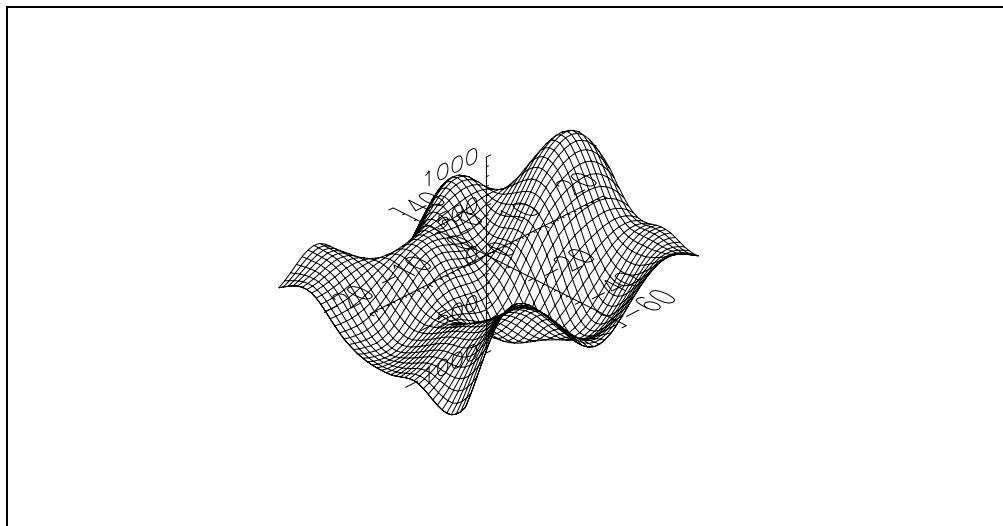


Figure 53: A surface displayed with the axes through the origin. The 3D space was created with the `Scale3` command.

Notice that the Y axis in this plot does not appear to extend to the edges of the plot. This is an optical illusion caused by the rotation of the plot. What can you do to convince yourself that this is really an illusion?

Before you move to the next section, be sure to restore your system variables to their original values. Type:

```
IDL> Restore, 'system.sav'
```

Combining Simple Graphical Displays

It is easy to take advantage of what you know so far about positioning graphics and setting up 3D coordinate systems to start to combine graphical displays in various ways. For example, it is often interesting to display an image together with a contour plot of the same data. To see how easy this is to do, open the 512 by 512 *Brain X-Ray* data set by typing this:

```
IDL> brain = LoadData(9)
```

We can load this into roughly 80 percent of the current graphics window with the `TVImage` command you downloaded to use with this book. But remember that this could distort the image if the current graphics window is not perfectly square. To preserve the aspect ratio of the image, and still try to fit the image into about 80 percent of the window, set the `Keep_Aspect_Ratio` keyword, like this:

```
IDL> Window, XSize=450, YSize=350  
IDL> LoadCT, 3  
IDL> thisPosition = [0.2, 0.2, 0.8, 0.8]  
IDL> TVImage, brain, Position=thisPosition, $  
      Keep_Aspect_Ratio=1
```

When the *Keep_Aspect_Ratio* is set with *TVImage*, the *Position* keyword becomes an output parameter. In other words, the variable *thisPosition* now holds the normalized position coordinates that describes the position of the image in the window. These are different from the coordinates you put in because the positions had to be changed to keep the aspect ratio of the image unchanged. You can see the new position coordinates by printing the variable, like this:

```
IDL> Print, thisPosition
```

You can use these new position coordinates to draw a contour plot of the data right on top of the image that is already in the window. Be sure to use the *NoErase* keyword to keep the contour plot from erasing the display the way it usually does. The *XStyle* and *YStyle* keywords are necessary to keep the axes from autoscaling. Type:

```
IDL> Contour, brain, XStyle=1, YStyle=1, NLevels=8, $  
Position=thisPosition, /NoErase
```

Your output should look similar to the illustration in Figure 54.

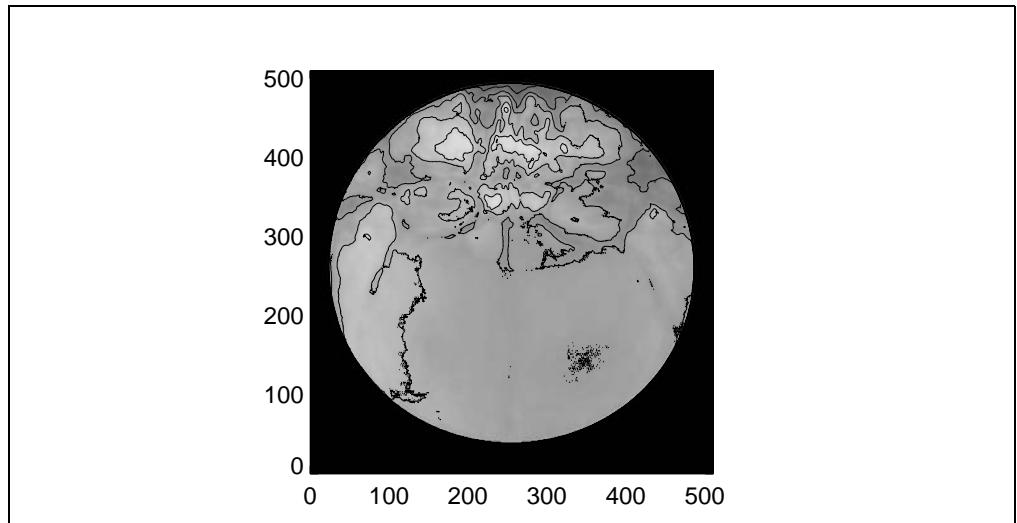


Figure 54: It is easy to combine images with contour plots in IDL.

It is also easy to combine surface and contour plots. Open the 41 by 41 *Elevation Data* data set with the *LoadData* command like this:

```
IDL> peak = LoadData(2)
```

You might want to be able to see, for example, a shaded surface rendition of this data at the same time that you look at a contour plot of the data. This is easily accomplished in IDL by using the *Shade_Surf* command to establish a 3D coordinate system, which is then used to position the contour plot. Your code might look like this:

```
IDL> Window, XSize=400, YSize=400  
IDL> TVLct, [70,0], [70,255], [70,0], 0  
IDL> LoadCT, 5, NColors=!D.Table_Size-2, Bottom=2  
IDL> !P.Charsize = 1.5  
IDL> Shade_Surf, peak, /Save, Background=0, Color=1, $  
Shades=BytScl(peak, Top=!D.Table_Size-2) + 2B
```

You might want to add another Z axis on the right-hand side of this plot, like this:

```
IDL> Axis, ZAxis=0, 40, 0, 0, /T3D, Color=1
```

Finally, you are ready to add the contour plot. Be sure to use the *NoErase* keyword to avoid erasing whatever is already on the display. The *ZValue* keyword positions the contour plots along the Z axis. The value given to the *ZValue* keyword will be in

normalized units. A value of 1.0 will put the contour plot on the top of the 3D coordinate space that has been created.

```
IDL> Contour, peak, NLevels=12, Color=1, ZValue=1.0, /T3D, $  
/NoErase, /Follow
```



Note that sometimes when graphics plots are located along the edges of a 3D coordinate space that some lines can get clipped. This is usually the result of round-off error in the calculation that places them in the 3D space. If some of the contour lines in your contour plot appear incomplete, add the *NoClip* keyword to the contour command, like this:

```
IDL> Contour, peak, NLevels=12, Color=1, ZValue=1.0, /T3D, $  
/NoErase, /Follow, /NoClip
```

Your output should look similar to the illustration in Figure 55.

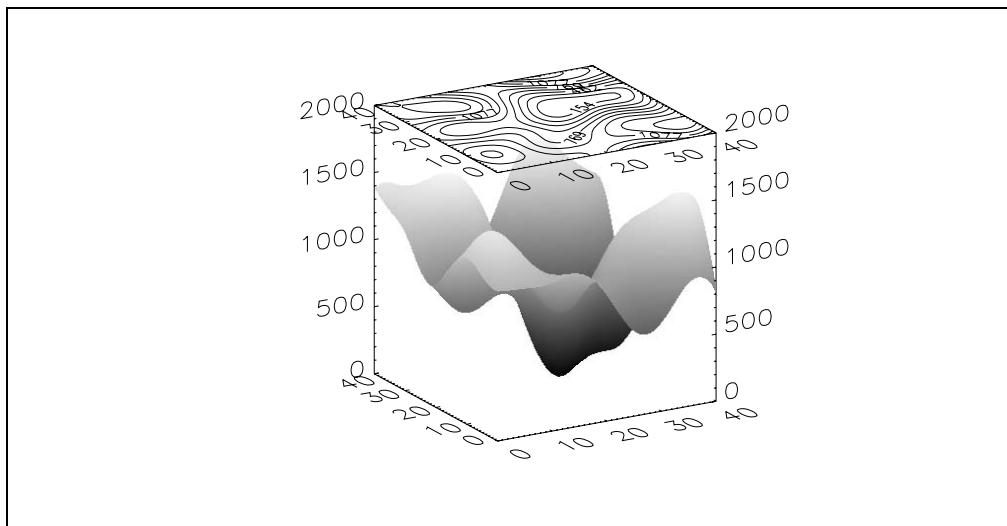


Figure 55: A combination shaded surface plot and contour plot.

Other combinations of graphics output commands are also possible. You are limited only by your imagination.

Animating Data in IDL

Another powerful graphical technique for visualizing your data is data animation. Quite often you can see information in an animation that is difficult or impossible to see looking at your data in other ways. If you don't have a 3D data set open, you can open the 80 by 100 by 57 *MRI Head* data set with the *LoadData* command, like this:

```
IDL> head = LoadData(8)
```

Animation is done in IDL with the *XInterAnimate* command. This command actually invokes an IDL widget program that is the main animation tool in IDL. *XInterAnimate* must be called three times. (1) Once to set up the animation tool and, in particular, the size of the animation frame buffer. (2) Once to load the animation tool. And, (3) one final time to run the animation sequence.

When a 3D animation array is available, *XInterAnimate* is particularly easy to set up and run. As an example, you will animate the *MRI Head* data set in the Z direction. That is, the animation will run from the bottom of the head to the top along the Z axis. At the moment each frame of data in the Z direction is a 80 by 100 array. This is a little small for a good animation, but you will deal with that problem in a moment.

Setting Up the Animation Tool

First, set up the animation tool. There will be 57 frames in this animation, with each frame an 80 by 100 image. The frame buffer, then, will be set to 80 by 100 by 57. It is set up like this. Notice the *Showload* keyword. This keyword is set so you can see the animation as it is loading. It is not necessary to show the user these steps if for some reason you don't want to. Type:

```
IDL> XInterAnimate, Set=[80,100,57], /Showload
```

Loading the Animation Buffer

The next step is to load the animation buffer with data. In this case, you are going to load the animation tool buffer from the 3D image data set you already have available. This command, which is always in a loop, looks like this:

```
IDL> FOR j=0, 56 DO $  
      XInteranimate, Frame=j, Image=head[*,*,j]
```

As this command executes you will see each frame loading into the animation buffer. This is *not* the animation.

Running the Animation Tool

The animation is run by typing the *XInteranimate* command one final time, like this:

```
IDL> XInteranimate
```

The animation tool should look similar to the illustration in Figure 56.

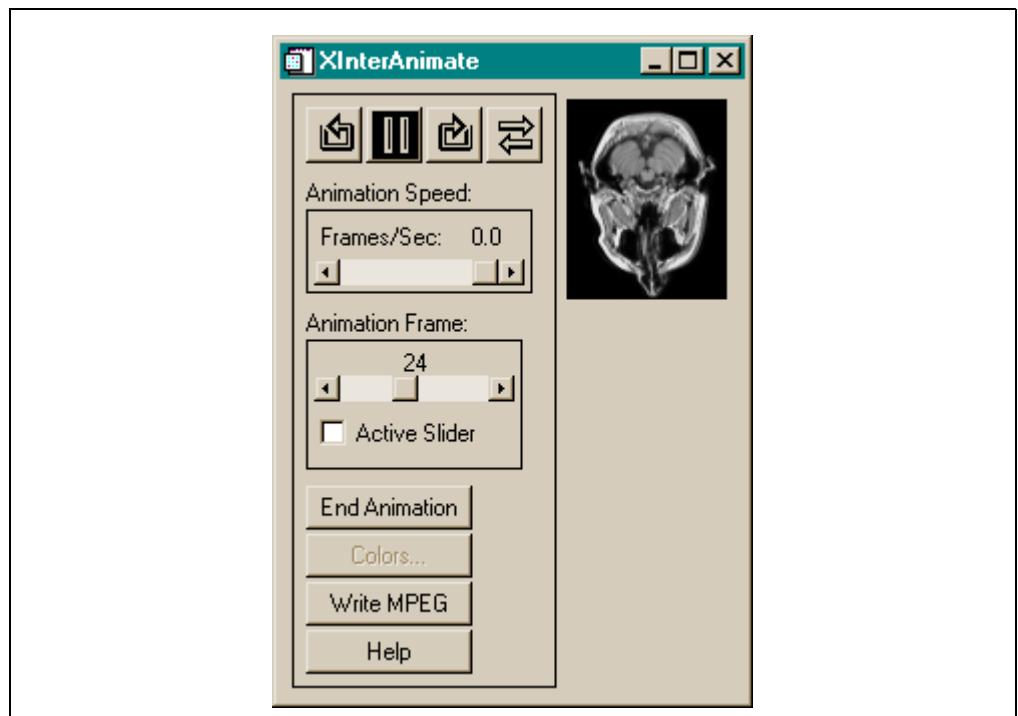


Figure 56: The *XInterAnimate* animation tool, with the MRI Head data set loaded.

Controlling the Animation

Experiment with some of the animation controls. You can determine how the animation frames loop, the speed of the animation, and you can even change color

tables while the animation is running. Note that if you click the *End Animation* button, you will have to type all three animation commands over again to get the animation working. You will have to stop the animation before you can use the slider to go to a particular animation frame.

Normally, animations run as fast as your machine allows them to run. (A widget timer event is responsible for getting the next animation frame in the sequence.) The animation speed button adds a delay to this timer event. If you want to start your animation out at a slower speed initially, you can give the final *XInterAnimate* command a speed parameter. This number will be between 0 and 100. For example, to start the animation off at 50% of its fastest speed, you might type this:

```
IDL> XInterAnimate, Set=[80,100,57], /Showload  
IDL> FOR j=0, 56 DO $  
      XInteranimate, Frame=j, Image=head[*,*,j]  
IDL> XInteranimate, 50
```

Saving Animation Pixmaps

One of the reasons the animation tool is as fast as it is, is that it uses pixmaps and the device copy technique to perform the animation. (See “The Device Copy Method of Erasing Annotation” on page 116 for additional information about this technique.) Normally, these pixmaps are deleted when you click the *End Animation* button. If you keep the pixmaps in memory, you can immediately start a new animation sequence at any time by just typing the word *XInterAnimate*. The way to keep the pixmaps in memory is to use the *Keep_Pixmaps* keyword when you start the animation, like this:

```
IDL> XInterAnimate, Set=[80,100,57], /Showload  
IDL> FOR j=0, 56 DO $  
      XInteranimate, Frame=j, Image=head[*,*,j]  
IDL> XInteranimate, 50, /Keep_Pixmaps
```

You can now exit the animation program and start it up again by just typing this:

```
IDL> XInterAnimate
```

You will want to be sure you delete the pixmaps when you are finished with them. You do that with the *Close* keyword, like this:

```
IDL> XInterAnimate, /Close
```

Animating Other Types of Graphic Data

Data that is in 3D arrays is not the only type of data that can be animated in the animation tool. In fact, many times what you want to animate is what you put into an IDL graphics window. The *XInterAnimate* tool has the ability to take a snap-shot of an IDL graphics window and store that as a frame in the animation buffer. This is done with the *Window* keyword rather than the *Image* keyword.

You can see how this is done with the data set you have open now. This little 80 by 100 image that you are displaying is quite small. You probably want it to be larger. But you probably don't want to create another, larger array to store the data. This would be an inefficient use of IDL memory. However, you could make each image larger by re-binning the image when it is displayed. This would take no additional memory in IDL.

Suppose you want each animation frame to be 256 by 320 and you would like to include the frame number on the animation. You might set up your animation code like this. The idea is to fill up the graphics window with whatever graphics we want to display, and load the animation loop pixmaps by copying the contents of the window. This is being done with the *Window* keyword. Type this code in a new editor window as you see it here.

```

XInterAnimate, Set=[256,320,57], /Showload
yellow = GetColor('yellow', !D.Table_Size-2)
LoadCT, 3, NColors=!D.Table_Size-2
FOR j=0, 56 DO BEGIN
    TVImage, BytScl(head[*,*,j], Top=!D.Table_Size-3)
    XYOutS, 0.1, 0.1, /Normal, StrTrim(j,2), Color=yellow
    XInteranimate, Frame=j, Window=!D.Window
ENDFOR
XInteranimate, 50
END

```

Save the program as *headanimate.pro*. To run this program (which is called a main-level IDL program), type this:

```
IDL> .Run headanimate
```

You can put *any* IDL graphics commands inside the loop. This gives you considerable flexibility with the kinds of data you can animate. You're output should look similar to the illustration in Figure 57.

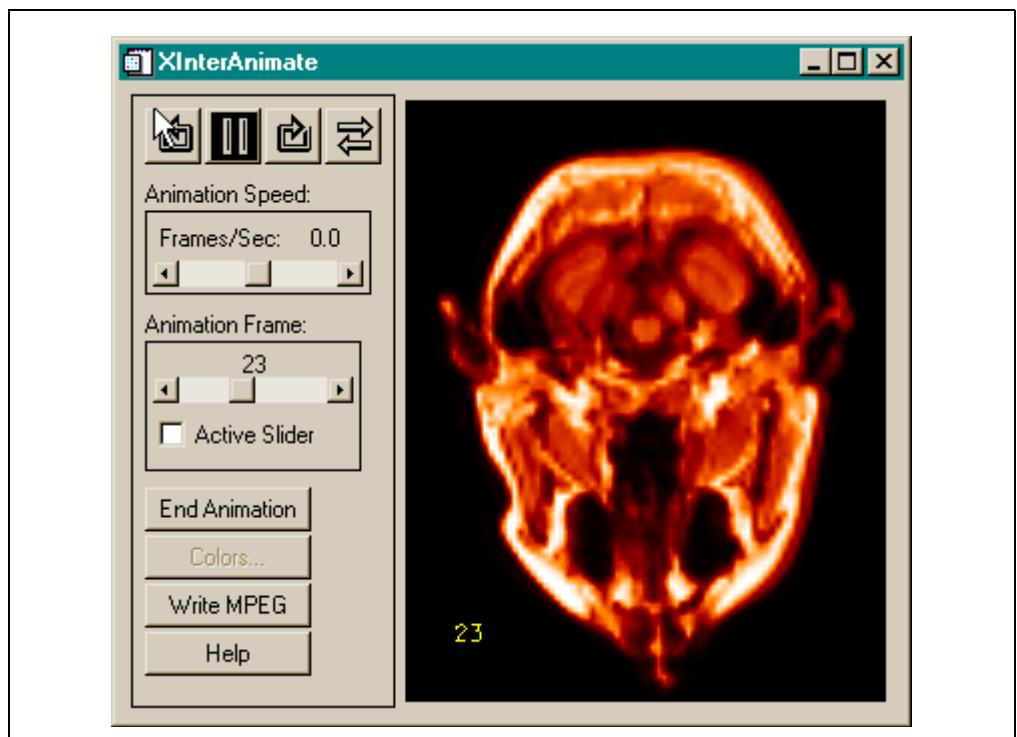


Figure 57: The animation pixmaps are loaded by filling the window with graphics and coping the contents of the window to the pixmap.

Notice the *Write MPEG* button on the *XInterAnimate* interface. Indeed, you can use the button to create MPEG movies of your data. But be careful. The button suggests that it is easier to do than it really is, at least in IDL version 5.3. In fact, you should be careful that your animation window is some multiple of 16 (as in the example above) or you might obtain strange results. And not all MPEG viewers can play the MPEG movies created by IDL. Be sure you have a recent version of the MPEG viewer that supports the “latest” MPEG standard.

Gridding Data for Graphical Display

Many of the IDL graphical display routines (e.g., *Surface*, *Contour*, *Shade_Surf*, etc.) require data to be arranged in a 2D gridded array. (The data doesn't have to be *regularly* gridded, generally.) But, occasionally, this is not the kind of data you have available to you. For example "station data" comes from randomly located collection points. This kind of data must be gridded before it can be displayed.

To load this kind of randomly distributed XYZ data to work with these exercises, use the *LoadData* command to load the *Randomly Distributed (XYZ)* data set. This data set contains three 41-element vectors representing station latitude and longitude coordinates, and a sampled value. Type:

```
IDL> data = LoadData(14)
IDL> lon = data[0,*]
IDL> lat = data[1,*]
IDL> value = data[2,*]
```

You can plot the latitude and longitude vectors on a grid so you can see they are in fact randomly distributed. Type:

```
IDL> LoadCT, 0
IDL> TVLCT, [70,255,0], [70,255,255], [70,0,0], 1
IDL> Window, XSize=400, YSize=400
IDL> Plot, lon, lat, /YNoZero, /NoData, Background=1
IDL> PlotS, lon, lat, PSym=5, Color=2, SymSize=1.5
```

Your output will look similar to the illustration in Figure 58.

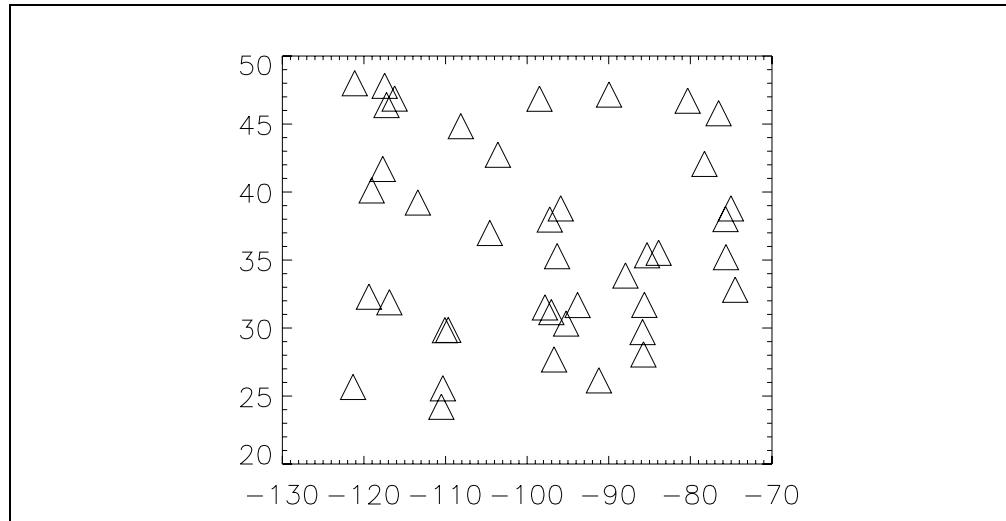


Figure 58: Randomly distributed latitude and longitude coordinates.

Delaunay Triangulation Method of Gridding

IDL uses a method of gridding data called the *Delaunay triangulation method*. This algorithm is not the best algorithm ever devised for gridding data, but it has the advantage that it is fast, relatively straightforward, and widely recognized as an acceptable gridding method. It depends upon creating a set of Delaunay triangles from the X and Y coordinates, such that no triangle contains the vertex of another triangle within its boundary. The Delaunay method involves interpolating regularly gridded values from the values associated with the vertices of the triangles.

The *Triangulate* command is used to create the Delaunay set of triangles. The inputs to the command are the XY coordinates of the triangle indices. The command looks

like this, where the variable *angles* is an output variable and will contain the set of Delaunay triangles returned from the calculation:

```
IDL> Triangulate, lon, lat, angles
```



Note that you can return the *convex hull* of these points with the *Triangulate* command, too. Simply add a fourth variable argument to the command above and the points on the perimeter of this set of points will be returned to you. The convex hull is useful in many arithmetic operations.

You can visualize this set of triangles (there are 70 triangles in this data set), by typing this command:

```
IDL> FOR j=0,69 DO BEGIN t = [angles[*,j], angles[0,j]] & $  
      Plots, lon[t], lat[t], Color=3 & ENDFOR
```

Your output will look like the illustration in Figure 59.

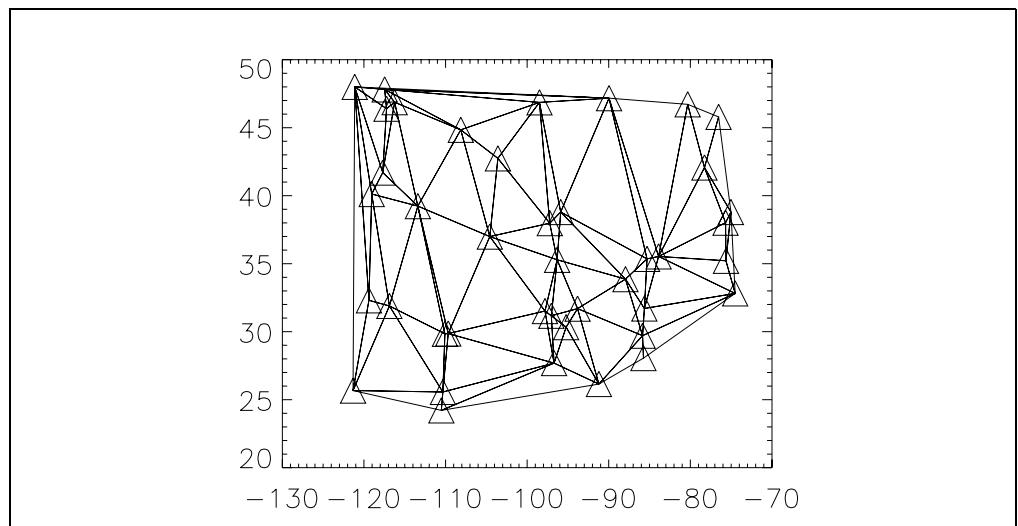


Figure 59: The set of Delaunay triangles returned by the *Triangulate* command.

To grid the data, take the set of triangles that came back from *Triangulate* and pass it to the *Trigrid* procedure. You can set up the grid endpoints or bounds, and the grid spacing. You can also specify the value for data that lies outside the triangles with the *Missing* keyword. Moreover, you can get the new latitude and longitude vectors that go with the gridded data from the output keywords *XGrid* and *YGrid*. For example, you can type this:

```
IDL> latMax = 50.0  
IDL> latMin = 20.0  
IDL> lonMax = -70.0  
IDL> lonMin = -130.0  
IDL> mapBounds = [lonMin, latMin, lonMax, latMax]  
IDL> mapSpacing = [0.5, 0.25]  
IDL> gridData = Trigrid(lon, lat, value, angles, $  
      mapSpacing, mapBounds, Missing=Min(value), $  
      XGrid=gridlon, YGrid=gridlat)
```

You can now use the gridded data in those IDL commands that require it.

```
IDL> Contour, gridData, gridlon, gridlat, /Follow, $  
      NLevels=10, XStyle=1, YStyle=1, Background=1, Color=2
```

Your output will look like the illustration in Figure 60.

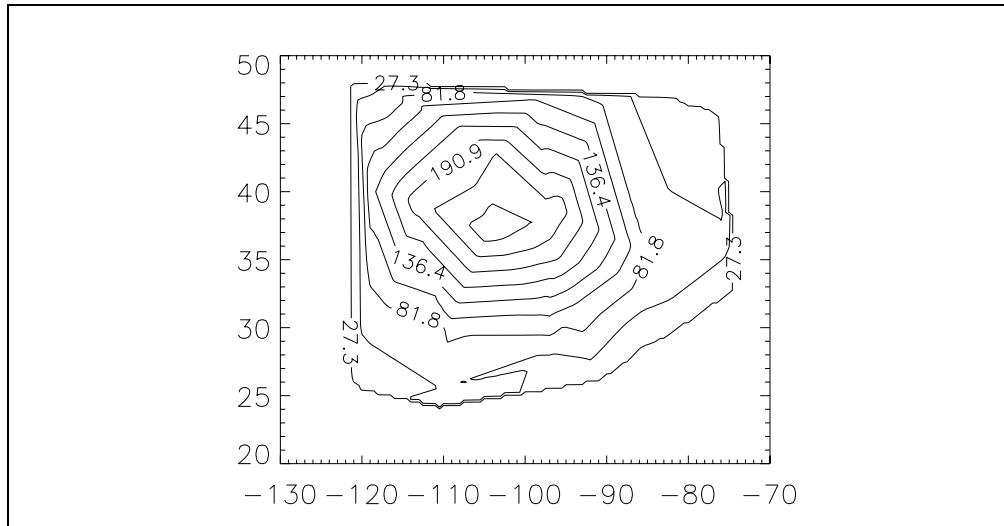


Figure 60: The results of the *TriGrid* command displayed as a contour plot.

Note that you can smooth the surface by using the *Quintic* or *Smooth* keywords (they are synonymous), like this:

```
IDL> gridData = Trigrid(lon, lat, value, angles, $  
mapSpacing, mapBounds, Missing=Min(value), $  
XGrid=gridlon, YGrid=gridlat, /Quintic)  
IDL> Contour, gridData, gridlon, gridlat, /Follow, $  
NLevels=10, XStyle=1, YStyle=1, Background=1, Color=2
```

You can also get better results sometimes by using the *Extrapolate* keyword to allow the extrapolation of values outside the triangle boundary. For example, you can type this:

```
IDL> Triangulate, lon, lat, angles, hull  
IDL> gridData = Trigrid(lon, lat, value, angles, $  
mapSpacing, mapBounds, Missing=Min(value), $  
Extrapolate=hull)  
IDL> Contour, gridData, gridlon, gridlat, /Follow, $  
NLevels=10, XStyle=1, YStyle=1, Background=1, Color=2
```

Your output should look similar to the illustration in Figure 61.

Spherical Gridding of Data

If this were really latitude and longitude data, you would not want to be gridding it on a flat surface. The surface of the Earth is more spherical. The gridding routines *Triangulate* and *TriGrid* also allow you to apply spherical gridding to your data. In this case, the triangles that are produced are spherical triangles.

To put this data on a map projection, type:

```
IDL> Map_Set, /Orthographic, /Grid, /Continents, /Label, $  
/Isotropic, 35, -100, Color=1  
IDL> PlotS, lon, lat, PSym=4, Color=2
```

To grid this data with spherical gridding many of the same parameters can be used as before, although this time different keywords are used in the commands. Be sure to set the *Degrees* keyword, or the commands will calculate the spherical triangles in radians. The commands might look like this:

```
IDL> Triangulate, lon, lat, Sphere=angles, $
```

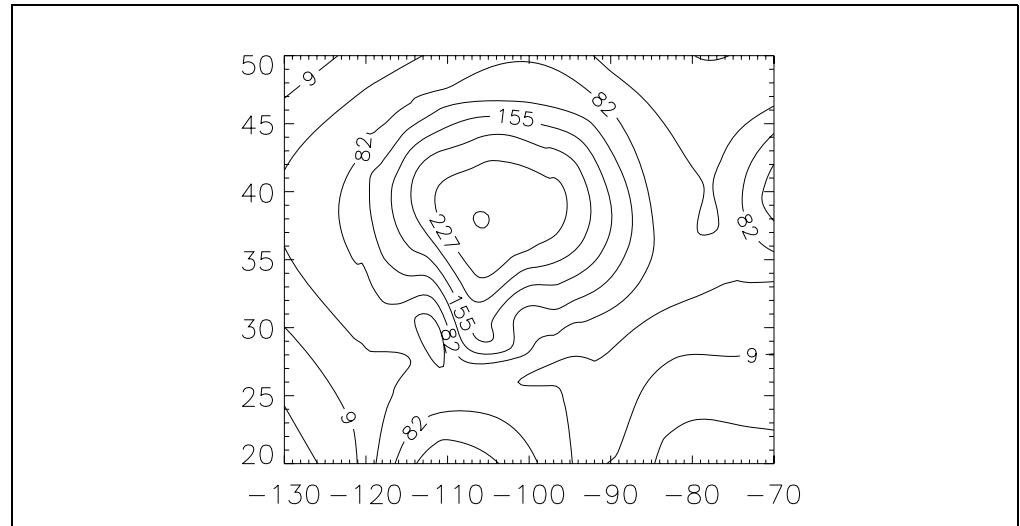


Figure 61: The results of *TriGrid* with the *Extrapolation* and *Smooth* keywords set.

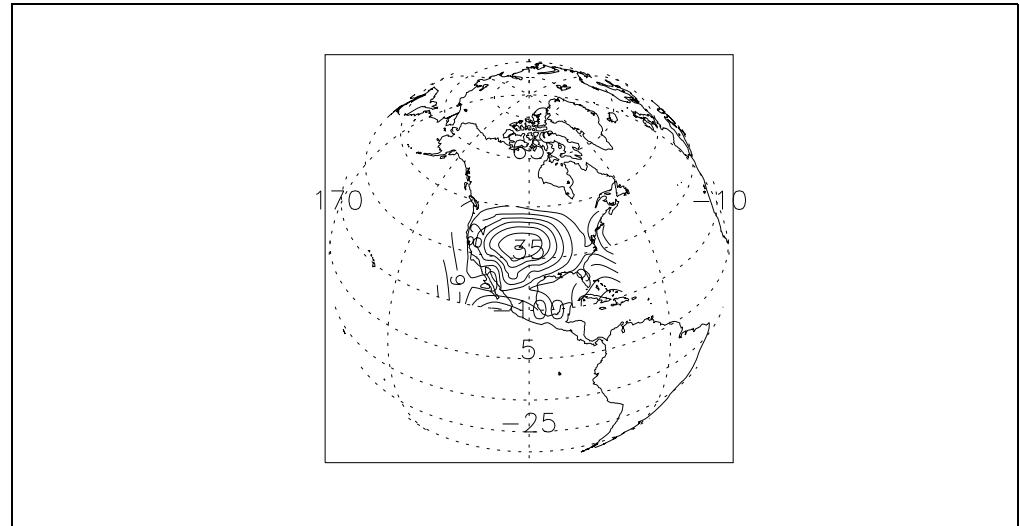


Figure 62: Spherically gridded data displayed as a contour plot on top of a map projection.

```
FValue=value, /Degrees
IDL> gridData = Trigrid(value, Sphere=angles, $
      mapSpacing, mapBounds, Missing=Min(value), $ 
      /Extrapolate, /Degrees)
IDL> Map_Set, /Orthographic, /Grid, /Continents, /Label, $ 
      /Isotropic, 35, -100, Color=1
IDL> Contour, gridData, gridlon, gridlat, /Follow, $ 
      NLevels=10, XStyle=1, YStyle=1, Color=2, /Overplot
```

Your output should look similar to the illustration in Figure 62.

Using the Cursor with Graphical Displays

One of the reasons data is displayed visually is so the user can interact with it in one way or another. One way users like to interact with data is to use the mouse cursor to

select or annotate portions of their data. This kind of interaction is easily accomplished in IDL using the *Cursor* command.

To see how the *Cursor* command works, load the *Time Series* data set with the *LoadData* command like this:

```
IDL> curve = LoadData(1)
```

Display the curve by typing these commands:

```
IDL> Window, XSize=400, YSize=400
IDL> LoadCT, 0
IDL> yellow = GetColor('yellow', 1)
IDL> Plot, curve
```

The *Cursor* command accepts two arguments. These must be variables in which the position of the cursor when a mouse button is pushed is recorded. The *Cursor* command requires that the cursor be located in the current graphics window. (This is the window pointed to by the system variable *!D.Window*.) For example, if you type this command, IDL will be waiting for you to move your cursor into the current graphics window (window index 0 if you typed the commands above) and click the mouse button down. When you do, IDL will put the position of the cursor into the variables *xLocation* and *yLocation*. Type:

```
IDL> Cursor, xLocation, yLocation
```

If you print these values out, you will see that the values are given in data coordinate space. That is the *xLocation* values will be between 0 and 100 and the *yLocation* values will be between 0 and 30. (At least they will be if you clicked the mouse inside the plot boundaries. What happens if you do not?) The *Cursor* command returns data coordinate positions by default.

```
IDL> Print, xLocation, yLocation
```

When Is the Cursor Position Returned?

It would appear from the commands above that the cursor position is returned when the mouse button is pushed down, but this is not always the case. In fact, when the *Cursor* command reports its position is determined by keywords to the *Cursor* command. These keywords are:

Change	The position is reported when there is a <i>change</i> in the cursor's position, or when the user <i>moves</i> the cursor.
Down	The position is returned when the mouse button is pushed <i>down</i> .
NoWait	The position is returned immediately when the <i>Cursor</i> command is executed. There is no delay or wait for mouse buttons. This keyword is sometimes used in loops when objects are being moved on the display.
Up	The position is returned not when the mouse button is clicked down, but when it comes <i>up</i> or is released.
Wait	The <i>Cursor</i> command <i>waits</i> for the button to be pressed to report its position. As long as the button is pressed down, this keyword causes the <i>Cursor</i> command to act as though it had been invoked with the <i>NoWait</i> keyword. This is the default behavior for the <i>Cursor</i> command.



Be careful to use the proper keyword with the *Cursor* command, especially if you are using the *Cursor* command in a loop. Users sometimes get into the habit of thinking that the default behavior for the *Cursor* command is to only report back when the cur-

sor is clicked *down*. It is not. The default behavior is to *wait* for a click then act as if a *no wait* were in effect. In a loop this difference can be critical.

Which Mouse Button Was Used with the Cursor?

In addition to setting the behavior of the cursor, you also sometimes want to know which mouse button was used to respond to the *Cursor* command. For example, you may want to do one thing if the right mouse button was used and something different if the left mouse button was used in response to the *Cursor* command.

You can determine which button was used with the *Cursor* command by examining the *Button* field of the *!Mouse* system variable. (Older versions of IDL used the value of the *!Err* system variable for this same purpose.) This field is an integer bit map. Valid values for the *Button* field and their meanings are as follows:

- !Mouse.Button = 0** No button has currently been used.
- !Mouse.Button = 1** The left mouse button was used with the *Cursor* command.
- !Mouse.Button = 2** The middle mouse button was used.
- !Mouse.Button = 4** The right mouse button was used.

Annotating Graphics Output with the Cursor

One way the *Cursor* command might be used is to allow the user to interactively place symbols on a line plot. For example, type the commands below exactly as they appear here. When you hit the final carriage return, click your mouse five times in the current graphics window. Five symbols will be placed in the window. (If you make a typing mistake in the code below, start again with the first line when you correct it.) Type:

```
IDL> FOR j=0, 4 DO BEGIN $
IDL> Cursor, xloc, yloc, /Down & $
IDL> PlotS, xloc, yloc, PSym=4, SymSize=2, $
      Color=yellow & ENDFOR
```

Drawing a Box

You might want to select a portion of the display and draw a box around it. Here are some commands to select the diagonal corners of a box with the *Cursor* command, draw the box (be sure to draw the box so that it includes a portion of the actual data), and zoom the plot into the box coordinates. First, draw the plot:

```
IDL> Plot, curve
```

Next, use the cursor to select one corner of the box you want to draw. You will want to be sure to click the cursor in the current graphics window. To make sure you know which one that is and that it is not hidden, type:

```
IDL> WShow
```

Now type the first *Cursor* command. Click somewhere inside the plot axes:

```
IDL> Cursor, x1, y1, /Down ; Select one corner of box.
```

Now type the second *Cursor* command. Click somewhere inside the plot axes:

```
IDL> Cursor, x2, y2, /Down ; Select diagonal corner of box.
```

The coordinates returned from the *Cursor* commands above are in *data* coordinate space. Draw the box like this:

```
IDL> PlotS, [x1,x1,x2,x2,x1], [y1,y2,y2,y1,y1], Color=yellow
```

Your output will look something like the illustration in Figure 63, although the actual box on your plot will depend on where you clicked in the window.

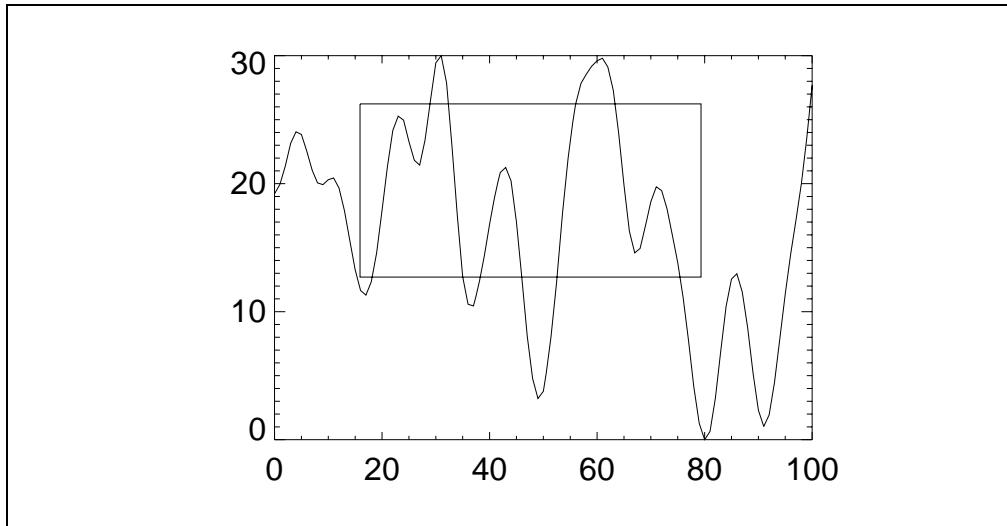


Figure 63: A line plot with a box drawn around the portion of the data. The box coordinates were selected with the *Cursor* command and the box drawn with the *PlotS* command.

To zoom into this portion of the plot, you will have to be sure the box coordinates are ordered properly. This is necessary because you may have first selected the lower-right corner of the box and then the upper-left, in which case $x1$ will be greater than $x2$. You can imagine other scenarios as well. To account for all of them, type:

```
IDL> xmin = Min([x1,x2], Max=xmax)
IDL> ymin = Min([y1,y2], Max=ymax)
```

Finally, you are ready to zoom into the portion of the data enclosed by the box. In addition to setting the data ranges properly, you also have to set the *[XY]Style* keywords. Do you know why? If you don't, try the command below without these two keywords. What happens?

```
IDL> Plot, curve, XRange=[xmin,xmax], YRange=[ymin,ymax], $
      XStyle=1, YStyle=1
```

Using the Cursor with Images

Normally when you are working with image data and using the *Cursor* command, you want the cursor locations in device coordinates rather than data coordinates. This is because there is usually a simple relationship (most of the time one-to-one) between the device coordinate and the equivalent location in the image. To see how this works, open the 360 by 360 *World Elevation* data set with the *LoadData* command, like this:

```
IDL> image = LoadData(7)
```

Display the image and load some colors like this:

```
IDL> topColor = !D.Table_Size-1
IDL> LoadCT, 3, NColors=!D.Table_Size-1
IDL> yellow = GetColor('yellow', topColor)
IDL> Window, XSize=360, YSize=360
IDL> TVImage, BytScl(image, Top=!D.Table_Size-2)
```

Now use the cursor to select a particular column and row in the image. Notice the *Device* keyword in the *Cursor* and *PlotS* commands. This is to be sure the coordinates are in *device* coordinates and not *data* coordinates. Draw a cross-hair at that location. (Be sure to click in the image window after you type the *Cursor* command.) Type:

```

IDL> s = Size(image)
IDL> Cursor, col, row, /Device ; Click in the window!
IDL> PlotS, [col, col], [0, s[2]], /Device, Color=yellow
IDL> PlotS, [0, s[1]], [row, row], /Device, Color=yellow

```

Notice how easy it is to access the data in the image in that particular column and row. For example, you can easily plot the column and row data profiles for the image, like this:

```

IDL> Window, 1, XSize=500, YSize=300
IDL> !P.Multi = [0, 2, 1]
IDL> Plot, image[*, row], Title='Row Profile'
IDL> Plot, image[col, *], Title='Column Profile'
IDL> !P.Multi = 0
IDL> WSet, 0

```

Your output should look similar to the illustration in Figure 64.

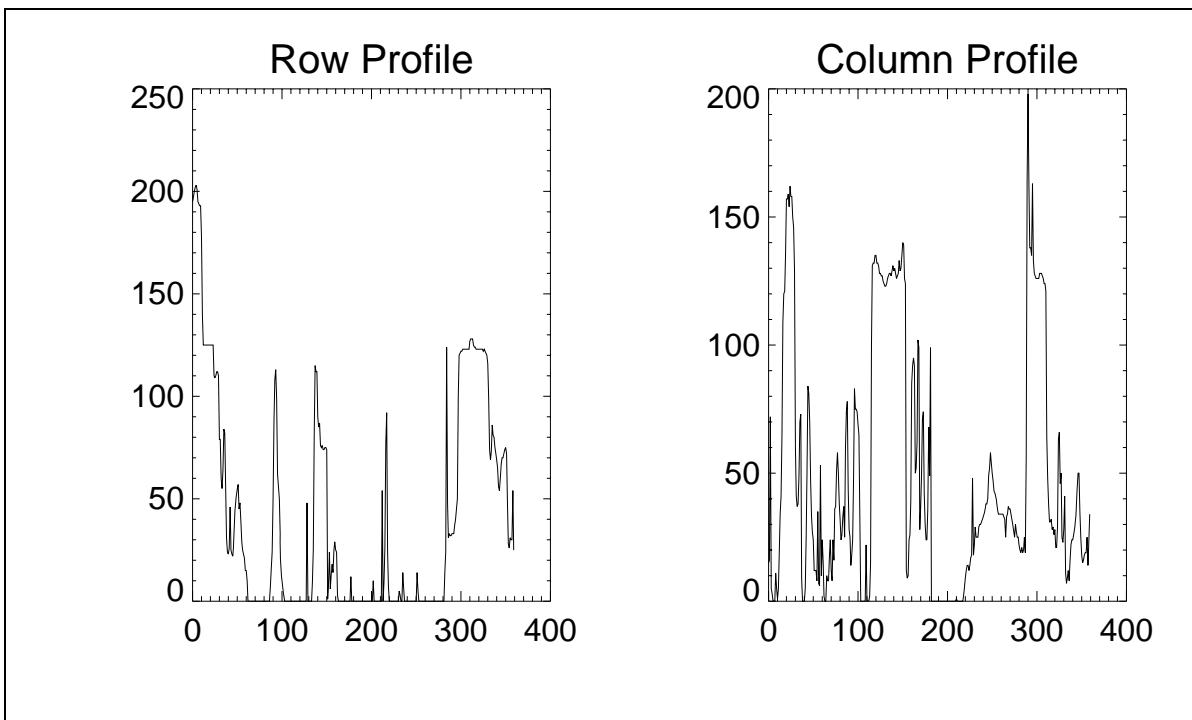


Figure 64: The column and row profiles of the image in the column and row selected with the *Cursor* command.

Using the Cursor in Loops

Sometimes you want to use the *Cursor* command in a loop. For example, you might want to know the value of each individual image pixel as you select it with the cursor. Here is a simple loop that continues until you click the *right* or *middle* mouse button to get out of it. Open a text editor and type these commands *exactly* as you see here.

```

yellow = GetColor('yellow', !D.Table_Size-2)
LoadCT, 3, NColors=!D.Table_Size-2
TVImage, BytScl(image, Top=!D.Table_Size-3)
!Mouse.Button = 1
REPEAT BEGIN
    Cursor, col, row, /Down, /Device
    Print, 'Pixel Value: ', image[col, row]
ENDREP UNTIL !Mouse.Button NE 1
END

```

Save the file as *loop1.pro*. (This file is among the files you downloaded to use with this book.) To compile and run this little main-level program, type:

```
IDL> .RUN loop1
```

Move your cursor into the image window and start clicking with the left mouse button. You will see the image pixel values printed out in your log window until you use some button other than the left in the image window.

What happens if you use other keywords besides *Down* with the *Cursor* command? Experiment a little and find out.

Erasing Annotation From the Display

Using the cursor to place annotations on the graphical display the way you have been doing tends to beg the question: "But, how do I *erase* what I just put there!" There are two preferred ways to erase annotations. I call these the *exclusive OR* method and the *device copy* method. Of the two, the device copy method gives more professional looking results, in my opinion. Both are presented here, but the focus will be on the device copy technique.

The "Exclusive OR" Method of Erasing Annotation

The exclusive OR method of erasing annotation works on the basis of what are called *graphics functions*. A graphics function is a bit-level operation on two numbers. These numbers are associated with the pixel that is already on the display (this is the so-called *destination* pixel) and the pixel you wish to put in that same location (this is the so-called *source* pixel).

Normally, the graphics function IDL uses is called *SOURCE*. In this graphics function, IDL ignores the value of the destination pixel and just puts the value of the source pixel at the pixel location. But if the graphics function is changed to *XOR* (exclusive OR) IDL does a bit-wise comparison of the bits of the destination pixel with the source pixel. This has the effect of "flipping" those bits of the destination pixel that are set in the source pixel. In other words, if the binary representations of the destination and source pixel are 01100101 and 11111111, respectively, then the binary representation of the destination pixel after the XOR operation is 10011010.

(The true XOR story is more complicated than this, because it only really works this way if IDL has 256 colors in contiguous locations in the color lookup table, and this is seldom the case. Most people just think of XOR mode as drawing in the "opposite" color and leave it at that. In true *XOR* mode you could predict what color you would be drawing with, but this is not true with this mode under most circumstances. This is the reason most professional IDL programmers prefer the device copy technique.)

The graphics function in effect at any particular time is set with the *Device* command and the *Set_Graphics_Function* keyword. *SOURCE* mode is graphics function 3; *XOR* mode is graphics function 6. At the moment IDL is in its default *SOURCE* mode. While you are in this mode, re-display the image in the image window. Type:

```
IDL> TVImage, BytScl(image, Top=!D.Table_Size-3)
```

Now, select *XOR* mode, like this:

```
IDL> Device, Set_Graphics_Function=6
```

You will draw a box on the image like this:

```
IDL> PlotS, [0.2, 0.2, 0.8, 0.8, 0.2], Color=yellow, $  
[0.2, 0.8, 0.8, 0.2, 0.2], /Normal
```

You will notice that the line of the box is not yellow, as you might have expected, but is instead a sort of multicolored hue, although it shows up reasonably well. The underlying pixels have been “flipped” in this graphics function.

To erase the box, all you have to do is flip the underlying pixel values back to their original values. This is easily done by issuing the *PlotS* command again, like this:

```
IDL> PlotS, [0.2, 0.2, 0.8, 0.8, 0.2], Color=yellow, $  
      [0.2, 0.8, 0.8, 0.2, 0.2], /Normal
```

You can issue the above command over and over, making the box appear and disappear at will. Before you go on, be sure you set your graphics function back to *SOURCE* mode, like this:

```
IDL> Device, Set_Graphics_Function=3
```

You can easily take advantage of graphics functions in your IDL programs. For example, open the *loop1.pro* main-level program you wrote earlier and modify it to look like this. Here you are going to draw a large cross-hair at each image location as you click in the image window. Save this program as *loop2.pro*. Type:

```
yellow = GetColor('yellow', !D.Table_Size-2)  
LoadCT, 3, NColors=!D.Table_Size-2  
Tvlct, 255, 255, 0, topColor  
TVImage, BytScl(image, Top=!D.Table_Size-3)  
!Mouse.Button = 1  
  
; Go into XOR mode.  
  
Device, Set_Graphics_Function=6  
; Get initial cursor location. Draw cross-hair.  
  
Cursor, col, row, /Device, /Down  
PlotS, [col,col], [0,360], /Device, Color=yellow  
PlotS, [0,360], [row,row], /Device, Color=yellow  
Print, 'Pixel Value: ', image(col, row)  
  
; Loop.  
  
REPEAT BEGIN  
    ; Get new cursor location.  
    Cursor, colnew, rownew, /Down, /Device  
    ; Erase old cross-hair.  
    PlotS, [col,col], [0,360], /Device, Color=yellow  
    PlotS, [0,360], [row,row], /Device, Color=yellow  
    Print, 'Pixel Value: ', image(colnew, rownew)  
    ; Draw new cross-hair.  
    PlotS, [colnew,colnew], [0,360], /Device, Color=yellow  
    PlotS, [0,360], [rrown,rrown], /Device, Color=yellow  
    ; Update coordinates.  
    col = colnew  
    row = rrown  
ENDREP UNTIL !Mouse.Button NE 1  
  
;Erase the final cross-hair.  
PlotS, [col,col], [0,360], /Device, Color=yellow  
PlotS, [0,360], [row,row], /Device, Color=yellow
```

```
; Restore normal graphics function.  
Device, Set_Graphics_Function=3  
END
```

Save the file as *loop2.pro*. (You can find *loop2.pro* among the files you downloaded to use with this book.) To compile and run this main-level program, type:

```
IDL> .RUN loop2
```

Place your cursor in the image window and click several times with your left mouse button. You should see a cross-hair at each cursor location. To exit the program, click the right or middle mouse button.

The Device Copy Method of Erasing Annotation

The device copy technique uses *pixmap windows* to erase annotation that you put on the display. A pixmap window is identical to any other IDL graphics window, except that it doesn't exist on your display. In fact, it exists in the video RAM of your display device. In other words, it exists in memory. But in every other respect, it is like a normal IDL graphics window: it is created with the *Window* command, it is made active with the *WSet* command, it is deleted with the *WDelete* command, etc. You draw graphics in a pixmap window in exactly the same way you draw graphics in a normal IDL graphics windows (e.g., with *Plot*, *Surface*, *TV*, and other graphics output commands).

The device copy technique involves copying a rectangular area from one window (called the *source window*) and pasting the rectangle into another window (called the *destination window*). The source and destination window can sometimes be the same window, as you will see in a moment. You see an illustration of the device copy technique in Figure 65.

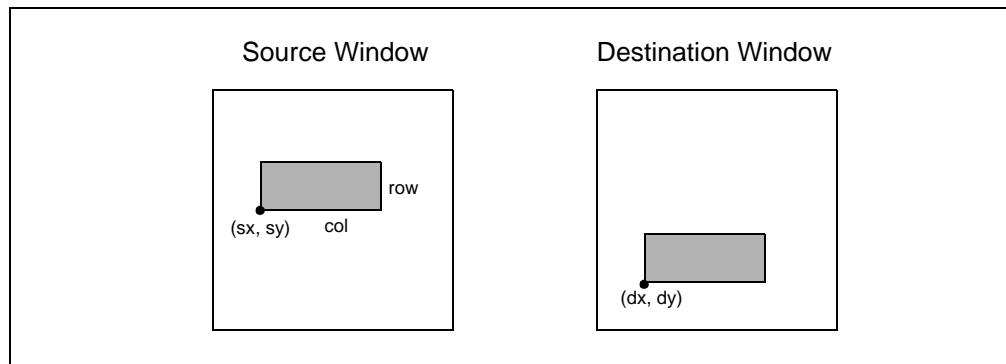


Figure 65: The device copy technique involves copying a rectangular portion of the source window into a location in the destination window. In practice entire windows may be copied or the source and destination window can be the same window.

The actual copying is done with the *Device* command and the *Copy* keyword (hence, the name of the technique). The general form of the command is this:

```
Device, Copy=[sx, sy, col, row, dx, dy, sourceWindowID]
```

In this command, the elements of the *Copy* keyword are:

sx, sy	The device coordinates of the lower-left corner of the rectangle in the <i>source</i> window. (The source window is the window the rectangle is being copied from.)
col	The number of columns to copy in the source window. This is the width of the rectangle.

row	The number of rows to copy in the source window. This is the height of the rectangle.
dx, dy	The device coordinates of the lower-left corner of the rectangle in the <i>destination</i> window. (The <i>destination</i> window is the window the rectangle is being copied to. The <i>destination</i> window is <i>always</i> the current graphics window.)
sourceWindowID	This is the window index number of the source window. The rectangle is copied from this window into the current graphics window (which is identified by the <i>!D.Window</i> system variable). The source window can be the current graphics window, but it is more often a window other than the current graphics window. It is often a pixmap window.

To see how this works, create a pixmap window and display the image in it. Pixmap windows are created with the *Window* command and the *Pixmap* keyword, like this:

```
IDL> Window, 1, /Pixmap, XSize=360, YSize=360
IDL> TVImage, BytScl(image, Top=!D.Table_Size-2)
```

Notice that you had no visual clue that anything happened when you typed these commands. This is because the pixmap window exists only in video RAM, not on the display. To be sure there is something in this window, open a third, regular window and try to copy the contents of the pixmap window into it. If your third window looks like your image window, you have typed the commands correctly. Type:

```
IDL> Window, 2, XSize=360, YSize=360
IDL> Device, Copy=[0, 0, 360, 360, 0, 0, 1]
```

Notice that you copied the entire contents of the pixmap window into this new window. This is similar to just re-displaying the image in the new window, except that it is several orders of magnitude faster. It is not unusual to copy the entire contents of a pixmap window into a display window, even if you just have to “repair” a portion of the display window.

Delete the last two windows you created (including the pixmap window), like this:

```
IDL> WDelete, 1, 2
```

It is important to remember to delete pixmap windows when you are finished with them. They do take up memory that you may want to use for something else. Some window managers allocate a fixed amount of memory for pixmap windows. Others use virtual memory if your pixmap window exceeds the capacity of the video RAM. X-terminals have notoriously little memory for pixmap windows.

To see how the device copy technique works in practice, modify the main-level program *loop2.pro* you wrote earlier. You may want to copy that program to another file and name it *loop3.pro*. Make the modifications shown below.

```
yellow = GetColor('yellow', !D.Table_Size-2)
LoadCT, 3, NColors=!D.Table_Size-2
TVImage, BytScl(image, Top=!D.Table_Size-3)
!Mouse.Button = 1

; Create a pixmap window and display image in it.

Window, 1, /Pixmap, XSize=360, YSize=360
TVImage, BytScl(image, Top=!D.Table_Size-3)

; Make the display window the current graphics window.

WSet, 0

; Get initial cursor location. Draw cross-hair.
```

```

Cursor, col, row, /Device, /Down
PlotS, [col,col], [0,360], /Device, Color=yellow
PlotS, [0,360], [row,row], /Device, Color=yellow
Print, 'Pixel Value: ', image[col, row]

; Loop.

REPEAT BEGIN

; Get new cursor location.

Cursor, colnew, rownew, /Down, /Device

; Erase old cross-hair.

Device, Copy=[0, 0, 360, 360, 0, 0, 1]
Print, 'Pixel Value: ', image(colnew, rownew)

; Draw new cross-hair.

PlotS, [colnew,colnew], [0,360], /Device, Color=yellow
PlotS, [0,360], [rrownew,rrownew], /Device, Color=yellow

ENDREP UNTIL !Mouse.Button NE 1

;Erase the final cross-hair and delete pixmap.

Device, Copy=[0, 0, 360, 360, 0, 0, 1]
WDelete, 1
END

```

Save the file as *loop3.pro*. (This file is among those you downloaded to use with this book.) To compile and run this main-level program, type:

```
IDL> .RUN loop3
```

Place your cursor in the image window and click several times with your left mouse button. To exit the program, click the right or middle mouse button. Notice that the cross-hairs are drawn in a yellow color.

Delete the pixmap window before you move on to the next exercise. Type:

```
IDL> WDelete, 1
```

Drawing a Rubberband Box

The device copy technique is an excellent one for drawing rubberband selection boxes and other shapes on the display. (A rubber band box is a box that has one fixed corner and one dynamic corner that follows the cursor around.) In fact, your *Loop3* program can be easily modified. Copy the *loop3.pro* program into a file named *rubberbox.pro*. (The *loop3.pro* file is among those you downloaded to use with this book.) Make the modifications below to see how easy it is to create a rubberband box.

```

yellow = GetColor('yellow', !D.Table_Size-2)
LoadCT, 3, NColors=!D.Table_Size-3
TVImage, BytScl(image, Top=!D.Table_Size-3)
!Mouse.Button = 1

; Create a pixmap window and display image in it.

Window, 1, /Pixmap, XSize=360, YSize=360
TVImage, BytScl(image, Top=!D.Table_Size-3)

; Make the display window the current graphics window.

WSet, 0
; Get initial cursor location (static corner of box).

```

```

Cursor, sx, sy, /Device, /Down
; Loop.

REPEAT BEGIN
    ; Get new cursor location (dynamic corner of box).

    Cursor, dx, dy, /Wait, /Device
    ; Erase the old box.

    Device, Copy=[0, 0, 360, 360, 0, 0, 1]
    ; Draw the new box.

    PlotS, [sx,sx,dx,dx,sx], [sy,dy,dy,sy,sy], /Device, $
        Color=yellow

ENDREP UNTIL !Mouse.Button NE 1
    ; Erase the final box and delete the pixmap.

Device, Copy=[0, 0, 360, 360, 0, 0, 1]
WDelete, 1
END

```

To run this program, type:

```
IDL> .RUN rubberbox
```

(This file is among those you downloaded to use with this book.)

Graphics Window Scrolling

Another good application of the device copy technique is to implement window scrolling. In this example you will scroll the image in the graphics display window with the device copy technique. The image will scroll four columns at a time from left to right. The algorithm you will use is this: (1) Copy the last four columns on the right of the window into a small pixmap window that is just four columns wide and 360 rows tall, then (2) Move the entire contents of the display window (minus the four columns you just copied) over to the right four columns in the same window (i.e., the source window and the destination window are identical), and finally (3) Copy the contents of the pixmap window into the first four columns on the left of the display window. Open a text editor and type the commands below. Name your program *scroll.pro*. (This program is among those you downloaded to use with this book.) Type:

```

; Open a pixmap window 4 columns wide.

Window, 1, /Pixmap, XSize=4, YSize=360
FOR j=0,360/4 DO BEGIN
    ; Copy four columns on right of display into pixmap.

    Device, Copy=[356, 0, 4, 360, 0, 0, 0]
    ; Make the display window the active window.

    WSet, 0
    ; Move window contents over 4 columns.

    Device, Copy=[0, 0, 356, 360, 4, 0, 0]
    ; Copy pixmap contents into display window on left.

    Device, Copy=[0, 0, 4, 360, 0, 0, 1]
ENDFOR

```

```
WDelete, 1  
END
```

To run this program, type:

```
IDL> .Run scroll
```

The program scrolls once. To run it again, type:

```
IDL> .Go
```

Can you modify the program to make it keep scrolling until you stop it?

Graphics Display Tricks in the Z-Graphics Buffer

You can think of the Z-graphics buffer in IDL as a three-dimensional box in which 3D objects can be deposited without regard to their “solidity.” The box has the ability to keep track of the “depth” of an object in a 16-bit depth buffer. One side of the box is a projection plane. You can think of rays of light going through each pixel in the projection plane and eventually encountering a solid object in the box. The pixel value the light ray encounters is the value that is “projected” onto the projection plane. In this way, the Z-graphics buffer can take care of hidden surface and line removal automatically. You see an illustration of this concept in Figure 66.

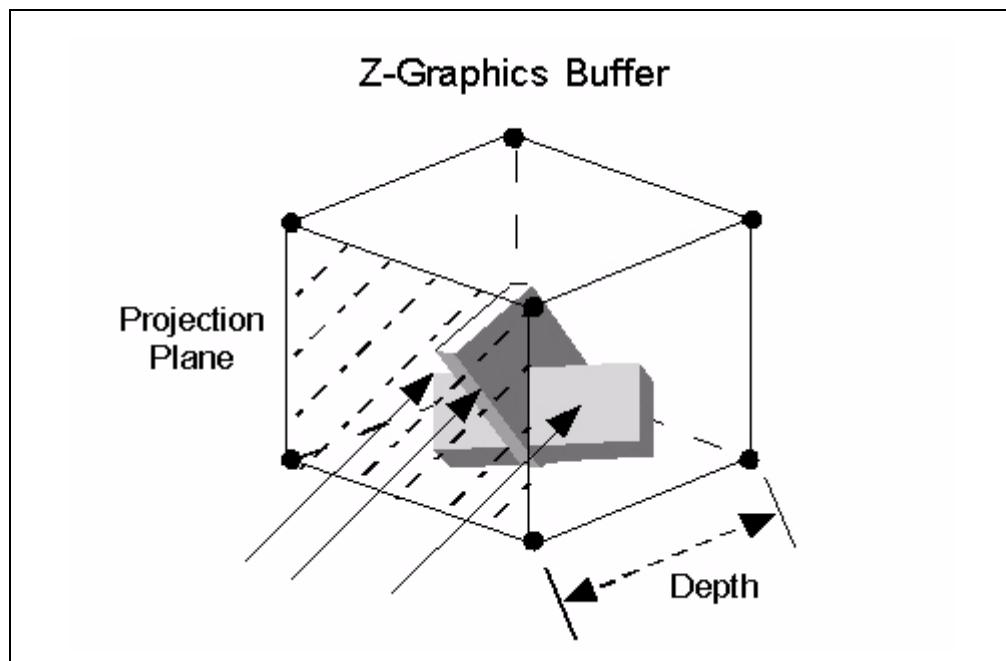


Figure 66: The Z-graphics buffer can be thought of as a 3D box that keeps track of depth information. Rays hit the objects in the Z-graphic buffer and their pixel values are projected back onto the projection plane.

The idea is that once you load your objects into the 3D box, you take a “snap-shot” or picture of the projection plane. This is the 2D projection of the 3D objects in the box. Objects that are behind other objects will not be shown. (This behavior can be modified by the *Transparent* keyword to some IDL graphics commands, as you will see.) The snap-shot is, in effect, a screen dump of the projection plane taken with the *TVRD* command.

The Z-Graphics Buffer Implementation

The Z-graphics buffer is implemented in software in IDL as another graphics output device, similar to the PostScript device or your normal X, Win, or Mac device. Thus, to write to the Z-graphics buffer you must make it the current graphics output device with the *Set_Plot* command. As with other graphics output devices, the Z-graphics buffer is configured with the *Device* command and appropriate keywords.

Two keywords that are often used with the Z-graphics buffer are *Set_Colors* and *Set_Resolution*. The keywords are defined like this:

Set_Colors	The number of colors in the Z-graphics buffer. By default the Z-graphics buffer uses 256 colors. This is seldom the number of colors in your IDL session. If you want the output from the Z-graphics buffer to have the same number of colors as your display device, you will need to set this keyword.
Set_Resolution	The projection plane of the Z-graphics buffer is normally set to 640 pixels wide and 480 pixels high. You should set the resolution of the Z-graphics buffer to the size of the graphics window you want to display the output in.

```
Device, Set_Resolution=[400, 400]
```

A Z-Graphics Buffer Example: Two Surfaces

To see how the Z-graphic buffer works, create two objects named *peak* and *saddle*, like this. (The commands to implement this example can be found in the file *twosurf.pro* that you downloaded to use with this book.)

```
IDL> peak = Shift(Dist(20, 16), 10, 8)
IDL> peak = Exp(- (peak / 5) ^ 2)
IDL> saddle = Shift(peak, 6, 0) + Shift(peak, -6, 0) / 2B
```

You are going to combine these two 3D objects in the Z-graphics buffer, but first you might like to see what these objects look like on their own. You will display them with different color tables in two windows. First, load a blue and red color table in different portions of the color lookup table. Type:

```
IDL> colors = !D.Table_Size/2
IDL> LoadCT, 1, NColors=colors
IDL> LoadCT, 3, NColors=colors, Bottom=colors-1
```

Create a window and display the shaded surface plot of the first object. Notice that the *Set_Shading* command is used to restrict the shading values to a particular portion of the color lookup table. Type:

```
IDL> Window, 1, XSize=300, YSize=300
IDL> Set_Shading, Values=[0,colors-1]
IDL> Device, Decomposed=0
IDL> Shade_Surf, peak, ZRange=[0.0, 1.2]
```

Display the second object in its own display window. Use a different portion of the color lookup table for the shading parameters. Type:

```
IDL> Window, 2, XSize=300, YSize=300
IDL> Set_Shading, Values=[colors, 2*colors-1]
IDL> Shade_Surf, saddle, ZRange=[0.0, 1.2]
```

Make the Z-Graphics Buffer the Current Device

To combine these two objects in the Z-graphics buffer, you must make the Z-graphics buffer the current graphics display device. This is done with the *Set_Plot* command.

The *Copy* keyword copies the current color table into the Z-buffer. Be sure to save the name of your current graphics display device, so you can get back to it easily. Type:

```
IDL> thisDevice = !D.Name  
IDL> Set_Plot, 'Z', /Copy
```

Configure the Z-Graphics Buffer

Next, you must configure the Z-graphics device to your specifications. In this case, you want to restrict the number of colors and you want to make the buffer resolution equivalent to the size of the current graphics display windows. Type:

```
IDL> Device, Set_Colors=2*colors, Set_Resolution=[300,300]
```

Load the Objects into the Z-Graphics Buffer

Now, put the two objects into the Z-graphics buffer. Notice that you don't see anything happening as you type these commands. The output is going into the Z-graphics buffer in memory, not to the display device. Type:

```
IDL> Set_Shading, Values=[0,colors-1]  
IDL> Shade_Surf, peak, ZRange=[0.0, 1.2]  
IDL> Set_Shading, Values=[colors, 2*colors-1]  
IDL> Shade_Surf, saddle, ZRange=[0.0, 1.2], /NoErase
```

Take a Picture of the Projection Plane

Next, take a "snap-shot" of the projection plane. This is done with the *TVRD* command, like this:

```
IDL> picture = TVRD()
```

Display the Result on the Display Device

Finally, return to your display device, open a new window to display the result, and display the "picture," like this:

```
IDL> Set_Plot, thisDevice  
IDL> Window, 3, XSize=300, YSize=300  
IDL> TV, picture
```

Your output should look similar to the illustration in Figure 67.

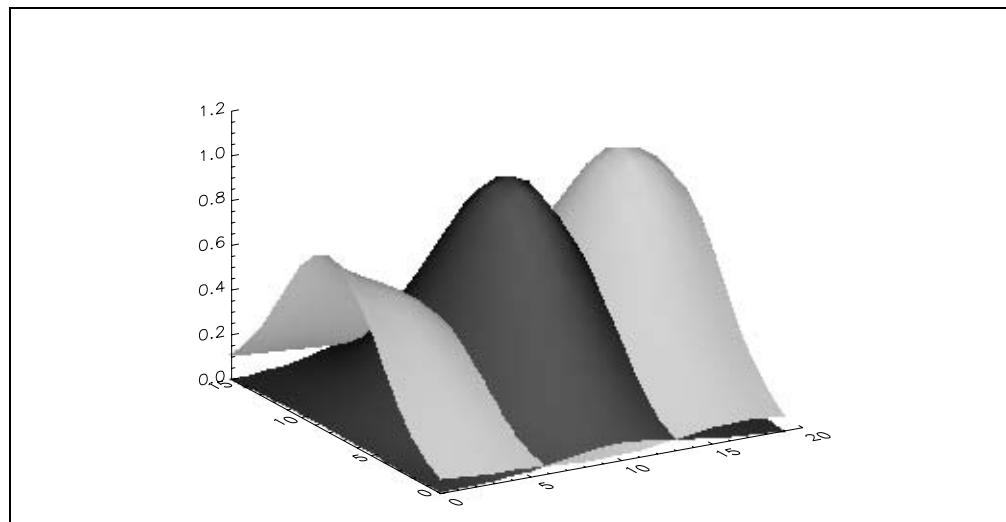


Figure 67: The Z-graphics buffer can be used to combine 3D objects with hidden surface removal performed automatically.

Some Z-Graphics Buffer Oddities

Look carefully at the output in, say, windows 1 and 3. Look particularly at the axes labels. Notice that the axes annotations are written slightly larger in window 3, the output that came from the Z-graphics buffer. The Z-graphics buffer for some reason uses a different default character size than IDL does when it is displaying graphics in a display window.

This simple fact can cause you untold hours of difficulty if you don't realize it when you are setting up a 3D coordinate space in the Z-graphics buffer and combining IDL graphics commands in the Z-graphics buffer. This is primarily because plot margins are based on default character size and plot margins are not the same on the display and in the Z-graphics buffer. They are *almost* the same. But it is the "almost" that will drive you crazy.

A rule of thumb that is *enormously* helpful, is to always set the *!P.Charsize* system variable if you are going to be doing graphics in the Z-graphics buffer. For example, like this:

```
IDL> !P.Charsize = 1.0
```

Just to give you an example, look at the illustration in Figure 67. The axes on this plot were not created in the Z-graphics buffer because then they would have been rendered in screen resolution (i.e., as an image) and I wanted to render them in PostScript resolution. If the *!P.Charsize* keyword had *not* been set prior to rendering the shaded surfaces and adding the axes later, it would have been impossible to get the axes to line up in the correct position in the final output.

Warping Images with the Z-Graphics Buffer

One of the more powerful techniques to use with the Z-graphics buffer is to use it to display slices of a 3D data set. This is possible because of the ability to warp images onto a polygon plane with the Z-graphics buffer. To see how this works, open the 80 by 100 by 57 3D *MRI Head Scan* data set with the *LoadData* command, like this:

```
IDL> head = LoadData(8)
```

You may want to open a journal file to capture these commands as you type them, since there are many of them and you have to get them exactly right. A journal file will enable you to make changes and re-run the commands easily. (This journal file has already been created for you and can be found in the file *warping.pro* that you downloaded to use with this book.)

```
IDL> Journal, 'warping.pro'
```

In general, the dimensions or size of a variable are found with the *Size* command in IDL. You need to know the sizes of the three dimensions in order to define the proper image plane. In this case, the "size" is going to be one less than the true size of the dimension, because you want to use this number as an index into an array, and IDL uses zero-based indexing. Type:

```
IDL> s = Size(head)
IDL> xs = s[1] - 1
IDL> ys = s[2] - 1
IDL> zs = s[3] - 1
```

Suppose you want to display the three orthogonal slices at the center of this data set, say through the 3D point (40, 50, 27). You can define these points, like this:

```
IDL> xpt = 40
IDL> ypt = 50
IDL> zpt = 27
```

Next, you want to construct the individual polygons that describe these three image slices or planes. In this case, each polygon will be a simple rectangle with four points (the corners of the rectangle). Each point in the rectangle will be described by an (x,y,z) triple. Another way to say this is that each plane will be a 3 by 4 array. You can type this:

```
IDL> xplane = [ [xpt, 0, 0], [xpt, 0, zs], [xpt, ys, zs], $  
    [xpt, ys, 0] ]  
IDL> yplane = [ [0, ypt, 0], [0, ypt, zs], [xs, ypt, zs], $  
    [xs, ypt, 0] ]  
IDL> zplane = [ [0, 0, zpt], [xs, 0, zpt], [xs, ys, zpt], $  
    [0, ys, zpt] ]
```

The next step is to get the image data that will correspond to each image plane. This is done easily by using array subscripts in IDL. Type:

```
IDL> ximage = head[xpt, *, *]  
IDL> yimage = head[*, ypt, *]  
IDL> zimage = head[*, *, zpt]
```

Notice that these images are all 3D images (one dimension is a 1). What you want are 2D images associated with each image plane, so you must reformat these 3D images into 2D images with the *Reform* command, as shown below. In this case, the *Reform* command reformats the image into an 80 by 100 by 1 image. When the final dimension in an array is 1, IDL discards it. The result here is an 80 by 100 image.

```
IDL> ximage = Reform(ximage)  
IDL> yimage = Reform(yimage)  
IDL> zimage = Reform(zimage)
```

To display these images properly, you want to make sure they are scaled properly into the number of colors on your display. It is important to scale them in relation to the entire data set. Scale the data and load colors like this:

```
IDL> minData = Min(head, Max=maxData)  
IDL> topColor = !D.Table_Size-2  
IDL> LoadCT, 5, NColors=!D.Table_Size-1  
IDL> TVLCT, 255, 255, 255, topColor+1  
IDL> Device, Decomposed=0  
IDL> ximage = BytScl(ximage, Top=topColor, Max=maxData, $  
    Min=minData)  
IDL> yimage = BytScl(yimage, Top=topColor, Max=maxData, $  
    Min=minData)  
IDL> zimage = BytScl(zimage, Top=topColor, Max=maxData, $  
    Min=minData)
```

Next, you are ready to set up the Z-graphics buffer. The *Erase* command will erase whatever may have been left in the buffer previously. In this case, you are erasing with a white color, to give more definition to your display. Type:

```
IDL> thisDevice = !D.Name  
IDL> Set_Plot, 'Z'  
IDL> Device, Set_Colors=topColor, Set_Resolution=[400,400]  
IDL> Erase, Color=topColor + 1
```

Set up the 3D coordinate space with the *Scale3* command. Here the axes will be labelled with the size of each dimension. Type:

```
IDL> Scale3, XRange=[0,xs], YRange=[0,ys], ZRange=[0,zs]
```

You are finally ready to render the slices in the Z-graphics buffer. You will use the *Polyfill* command for this purpose. The *Pattern* keyword will be set to the image slice you wish to display. The *Image_Coord* keyword contains a list of the image

coordinates associated with each vertex of the polygon. The *Image_Interp* keyword specifies that bilinear interpolation occurs rather than nearest neighbor re-sampling as the image is warped into the polygon. The *T3D* keyword ensures that the polygon is represented in 3D space by applying the 3D transformation matrix to the final output. Type:

```
IDL> Polyfill, xplane, /T3D, Pattern=ximage, /Image_Interp, $  
      Image_Coord=[ [0,0], [0, zs], [ys, zs], [ys, 0] ]  
IDL> Polyfill, yplane, /T3D, Pattern=yimage, /Image_Interp, $  
      Image_Coord=[ [0,0], [0, zs], [xs, zs], [xs, 0] ]  
IDL> Polyfill, zplane, /T3D, Pattern=zimage, /Image_Interp, $  
      Image_Coord=[ [0,0], [xs, 0], [xs, ys], [0, ys] ]
```

Finally, take a snap-shot of the projection plane, and display the result, like this:

```
IDL> picture = TVRD()  
IDL> Set_Plot, thisDevice  
IDL> Window, XSize=400, YSize=400  
IDL> TV, picture
```

If you opened a journal file, close it now:

```
IDL> Journal
```

Your output should look like the illustration in Figure 68. If it doesn't, modify the code in your journal file with a text editor to fix the problem. To re-run the code, save the file and type this:

```
IDL> @warping
```

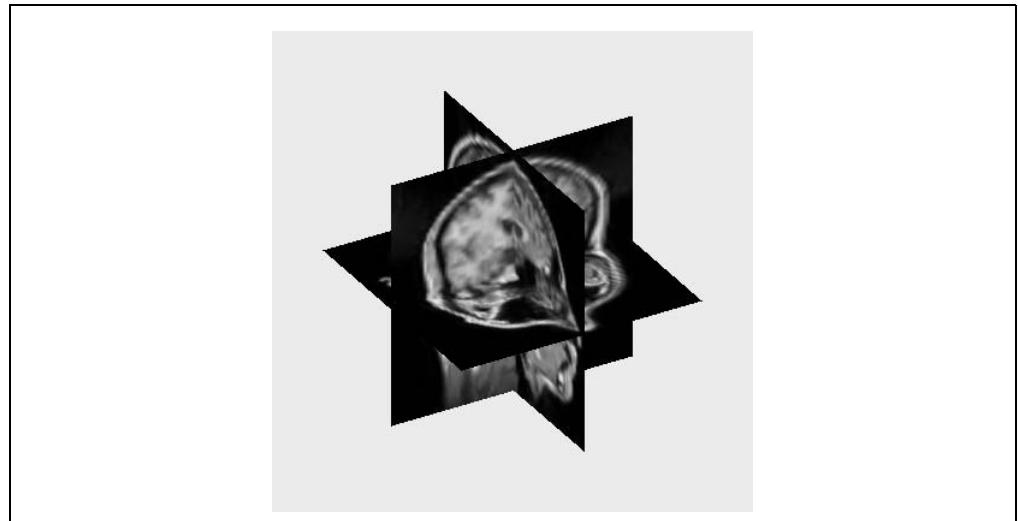


Figure 68: An example of warping image data into planes in the Z-graphics buffer.

Transparency Effects in the Z-Graphics Buffer

Notice that each slice in Figure 68 has quite a lot of black around the outside edges of the slice. This is not part of the image. Rather, it is part of the background noise.

One of the nice features of the Z-graphics buffer is that you can apply transparency effects in it. For example, if you set the *Transparent* keyword on the *Polyfill* command above to roughly 20 or 25, then all those pixels below that value in the image will be transparent. You can see what this looks like by typing this. (You may want to start another journal file. If you closed the last journal file, give this file a new name. Unfortunately, it is not possible to append to a journal file. Call the new journal file

transparent.pro. You can find a copy of this journal file among the files you downloaded to use with this book.)

```
IDL> Journal, 'transparent'
IDL> Set_Plot, 'Z'
IDL> Erase, Color=topColor + 1
IDL> Polyfill, xplane, /T3D, Pattern=ximage, /Image_Interp, $
    Image_Coord=[ [0,0], [0, zs], [ys, zs], [ys, 0] ], $
    Transparent=25
IDL> Polyfill, yplane, /T3D, Pattern=yimage, /Image_Interp, $
    Image_Coord=[ [0,0], [0, zs], [xs, zs], [xs, 0] ], $
    Transparent=25
IDL> Polyfill, zplane, /T3D, Pattern=zimage, /Image_Interp, $
    Image_Coord=[ [0,0], [xs, 0], [xs, ys], [0, ys] ], $
    Transparent=25
IDL> picture = TVRD()
IDL> Set_Plot, thisDevice
IDL> Window, /Free, XSize=400, YSize=400
IDL> Erase, Color=topColor + 1
IDL> TV, picture
IDL> Journal
```

If you make a mistake and the final output image doesn't look correct, correct the mistake in the journal file and re-run the journal file by typing this:

```
IDL> @transparent
```

Combining Z-Graphics Buffer Effects with Volume Rendering

Z-graphics buffer effects are often combined with volume rendering techniques to make vivid visual displays of data. For example, suppose you wanted to create an iso-surface of this data set. (An iso-surface is a surface that has the same value everywhere. It is like a three-dimensional contour plot of the data.) Start a journal file named *isosurface.pro*. (A copy of this journal file is among the files you downloaded to use with this book.)

```
IDL> Journal, 'isosurface.pro'
```

To create an iso-surface, first use the *Shade_Volume* command to create a list of vertices and polygons that describe the surface. In the command below, the *Low* keyword is set so that all the values greater than the iso-surface value will be enclosed by the iso-surface. The variables *vertices* and *polygons* are output variables. They will be used in the subsequent *PolyShade* command to render the surface. A look at the histogram of the head data suggests a value of 50 will be a suitable contouring level. Type:

```
IDL> Plot, Histogram(head), Max_Value=5000
IDL> Shade_Volume, head, 50, vertices, polygons, /Low
```

The iso-surface is rendered with the *PolyShade* command. Be sure to use the 3D transformation that you set up earlier, like this:

```
IDL> Scale3, XRange=[0, xs], YRange=[0, ys], ZRange=[0, zs]
IDL> isosurface = PolyShade(vertices, polygons, /T3D)
IDL> LoadCT, 0, NColors=topColor+1
IDL> TV, isosurface
```

Now, combine this iso-surface with the Z slice through the data set that you took earlier in the Z-graphics buffer. Notice that you are truncating the head data in the Z direction. Type:

```
IDL> Shade_Volume, head(*, *, 0:zpt), 50, vertices, $
    polygons, /Low
```

```
IDL> isosurface = PolyShade(vertices, polygons, /T3D)
IDL> isosurface(Where(isosurface EQ 0)) = topColor+1
IDL> TvLCT, 70, 70, 70, topColor+1
IDL> Set_Plot, 'Z', /Copy
IDL> TV, isosurface
IDL> Scale3, XRange=[0, xs], YRange=[0, ys], ZRange=[0, zs]
IDL> Polyfill, zplane, /T3D, Pattern=zimage, /Image_Interp, $
    Image_Coord=[ [0,0], [xs, 0], [xs, ys], [0, ys] ], $
    Transparent=25
IDL> picture = TVRD()
IDL> Set_Plot, thisDevice
IDL> TV, picture
IDL> Journal
```

Your output should look like the illustration in Figure 69. If it doesn't, modify your journal file and run it over again by typing this:

```
IDL> @isosurface
```



Figure 69: The isosurface and data slice combined in the Z-graphics buffer.

Chapter 5

◆ Discovering the Possibilities ◆◆◆



Reading and Writing Data in IDL

Chapter Overview

The purpose of this chapter is to introduce you to the general input and output routines in IDL. The basic rule in IDL is this: “If you have data, you can read it into IDL.” There is no IDL format, no special data massaging you have to do to make your data ready to be brought into IDL. This makes IDL one of the most powerful and flexible scientific visualization and analysis programs available.

Specifically, you will learn:

- How to open a file for reading and writing
- How to locate files and assign logical unit numbers
- How to obtain machine independent file names
- How to read and write ASCII or formatted data
- How to read and write unformatted or binary data
- How to work with large data files
- How to read and write files in popular file formats like GIF and JPEG

Opening a File for Reading or Writing

All input and output in IDL is done on logical unit numbers. You can think of a logical unit number as a pipe or conduit that is connected between IDL and the data file that you want to read from or write to. To read or write data from a file, you must first attach a logical unit number to a specific file. This is the purpose of the three *Open* commands in IDL:

<i>OpenR</i>	<i>Open</i> a file for <i>Reading</i> .
<i>OpenW</i>	<i>Open</i> a file for <i>Writing</i> .
<i>OpenU</i>	<i>Open</i> a file for <i>Updating</i> (i.e., reading and/or writing)

The syntax of these three commands is exactly the same: the name of the command, followed by a logical unit number and the name of the file to attach to that logical unit number. For example, to attach a file named *temp596.dat* to logical unit number 20 so you can write to it, you might type this:

```
OpenW, 20, 'temp596.dat'
```

More often you see *Open* commands written in a more general way. For example, you might see IDL code that looks like this:

```
OpenR, lun, filename
```

In this case, the variable *lun* holds a valid logical unit number and the variable *filename* holds a machine specific file name, which will be attached to the logical unit number.



Note that the *filename* variable name is machine *specific*. This means that it must be expressed with the local machine syntax if it contains directory specifications and it may be case sensitive on those machines (e.g., UNIX machines) in which file names are case sensitive.

Locating and Selecting Data Files

One of the reasons IDL is widely used is because it runs on many different computer operating systems. But because different operating systems have different file naming conventions (and, in particular, different ways of specifying subdirectories), this also presents a challenge in specifying file names in a machine independent fashion. Fortunately, IDL provides some tools to make this job easier for you.

Selecting File Names

Perhaps the easiest way to get a machine independent file name is to use the *Pickfile* dialog. This IDL command allows you to interactively pick a file name from a list of file names using the machine's native graphical file selection dialog. For example, to pick a file name from a list of the *.pro* files in your local directory you can type this command:

```
IDL> filename = Dialog_Pickfile(Filter='*.pro', /Read)
```



Note that this command was named *Pickfile* in versions of IDL prior to version 5.

IDL 5.2 introduced the ability to use *Dialog_Pickfile* to pick multiple files (if they reside in the same directory) by using the *Multiple* keyword. Use the normal platform-dependent way of selecting files. For example, on Windows machines select the first file normally, then use either a *Shift-Click* to select all the files between the first file and the second file or a *Control-Click* to select an additional file.

```
IDL> filename = Dialog_Pickfile(Filter='*.pro', /Read, $  
/Multiple)
```

If you want to open the file for writing instead of reading, use the *Write* keyword instead of the *Read* keyword in the dialog. You might even want to suggest a default filename, like this:

```
IDL> outfile = Dialog_Pickfile(File='default.dat', /Write)
```

What is returned from this dialog is the absolute path to the selected file in the form the machine IDL is running on expects. That is to say, it uses the native file name syntax. You can see this is so by typing this command:

```
IDL> Help, filename, outfile
```

Notice that the *Dialog_Pickfile* dialog has a *Cancel* button. If the *Cancel* button is selected, the dialog returns a null string. You always want to check to see if the name that comes back is null before you proceed to open the file for reading or writing.

```
IDL> IF outfile EQ '' THEN Print, 'Whoops!'
```

Selecting Directory Names

Dialog_Pickfile was improved in IDL 5.1 so that it can also be used to select a directory name rather than a file name. Set the *Directory* keyword so that only directories and not files are listed in the selection window:

```
IDL> directory = Dialog_Pickfile(/Directory)
```

Finding Files

Another useful command is the *FindFile* command. This command returns a string array containing the names of all files matching a given file specification. This is useful in automating file opening tasks from within IDL programs or for building a list of files from a directory without knowing how many files will be located there at any one time. For example, if you wanted to print the size (in bytes) of all the data files in the current directory, you might write IDL code like this:

```
files = FindFile('*.dat', Count=numfiles)
IF numfiles EQ 0 THEN Message, 'No data files here!'
FOR j=0,numfiles-1 DO BEGIN
    OpenR, lun, files(j), /Get_Lun
    fileInfo = FStat(lun)
    Print, fileInfo.size
    Free_Lun, lun
ENDFOR
```



One important point to note about the *FindFile* command is that the file specification (e.g., ‘*.dat’ above) is given in relative path names, not absolute path names. The command also returns relative path names and not absolute path names. This is different from the *Pickfile* dialog, which always returns an absolute path name.

Constructing File Names

A third useful IDL command for getting a machine independent file name is the *Filepath* command. For example, suppose you wanted to open the file *galaxy.dat*, which is in the *examples/data* subdirectory of the main IDL directory. You can construct a machine independent absolute path name to that file by typing this:

```
IDL> galaxy = Filepath('galaxy.dat', $
    Subdirectory=['examples','data'])
```

If you want to start in some other directory other than the main IDL directory, you can use the *Root_Dir* keyword to specify name of the starting directory. For example, to construct a path name to the same file in the *coyote* subdirectory of your current directory, you might type this:

```
IDL> CD, Current=thisDir
IDL> galaxy = Filepath('galaxy.dat', Root_Dir=thisDir, $
    Subdirectory='coyote')
```



Note that the *Filepath* command doesn’t actually *find* the file, it just constructs a file path name. The name can be constructed even if the file doesn’t exist on the machine.

Obtaining a Logical Unit Number

In IDL all file input and output is done on a logical unit number. The purpose of an *Open* command is to attach a specific file—indicated by its file name—with a logical unit number. There are 128 logical unit numbers available for you to use. These are separated into two categories.

Logical Unit Numbers	Purposes
1-99	These numbers can be used directly in <i>Open</i> commands
100-128	These numbers are obtained and managed by the <i>Get_Lun</i> and <i>Free_Lun</i> commands.

Table 9: *The 128 logical unit numbers are separated into two groups. One group you use directly. The other you manage with the *Get_Lun* and *Free_Lun* commands.*

Using Logical Unit Numbers Directly

To use a logical unit number directly, you just choose a number within the range of 1-99 and use it with an *Open* command. You are responsible for choosing a number that is not currently in use. For example, you could open the *galaxy.dat* file in the *coyote* subdirectory of your current directory on logical unit number 5 by typing this:

```
IDL> CD, Current=thisDir
IDL> filename = Filepath(Root_Dir=thisDir, $
   Subdirectory='coyote', 'galaxy.dat')
IDL> OpenR, 5, filename
```

Once a logical unit number in the range of 1-99 is assigned to a file, it cannot be reassigned until the logical unit number is closed or you exit IDL (which automatically closes all open logical unit numbers).

When you are finished with the logical unit number (i.e., you don't want to read or write to that file anymore) you close it and make it available to be used again with the *Close* command:

```
IDL> Close, 5
```

Allowing IDL to Manage Logical Unit Numbers

Most of the time (and especially in IDL programs) it is better to let IDL manage the logical unit numbers. The commands *Get_Lun* and *Free_Lun* are used for this purpose. There are two ways to let IDL obtain a logical unit number

You can use the *Get_Lun* command directly, like this:

```
IDL> Get_Lun, lun
IDL> OpenR, lun, filename
```

Or, you can do it indirectly, with the *Get_Lun* keyword to the *Open* command, like this:

```
IDL> OpenR, lun, filename, /Get_Lun
```

This command performs an implicit *Get_Lun* command and puts the resulting logical unit number in the variable *lun*. This is the way the logical unit number is most often obtained, especially in IDL programs. (Note that the variable name does not have to be *lun*. You can give it any name you like. If you open several files, you will want several different names.)

When you are finished with the logical unit number (i.e., you don't want to read or write to that file anymore) you close it with the *Free_Lun* command, like this:

```
IDL> Free_Lun, lun
```

The advantage of using the *Get_Lun* and *Free_Lun* commands is that you don't have to keep track of which logical unit numbers are available or unused. The *Get_Lun* procedure is guaranteed to return a valid logical unit number (assuming you have

opened fewer than 28 files all at once). If you open files from procedures and functions, it is an excellent idea to always use the *Get_Lun* command, since you can never guarantee that a particular logical unit number will be available if you select it directly.

Determining Which Files are Attached to Which LUNs

You can easily determine which files are attached to which logical unit number by using the *Help* command with the *Files* keyword, like this:

```
IDL> Help, /Files
```

Reading and Writing Formatted Data

IDL distinguishes between two types of formatted files with regard to reading and writing data: a free file format and an explicit file format. Formatted files are sometimes called ASCII files or plain text files.

Free File Format

A free format file uses either commas or whitespace (tabs and spaces) to distinguish each element in the file. It is a more informal format than an explicit file format.

Explicit File Format

An explicit format file is formatted according to rules specified in a format statement. The IDL format statement is similar to format statements you might write in a FORTRAN or C program.

Writing a Free Format File

Writing a free format file in IDL is extremely easy. You simply use the *PrintF* command to write variables into the file. This is exactly equivalent to using the *Print* command to write variables to the display. IDL automatically puts white space between each piece of data written into the file.

For example, type these commands to create some data to write into a file:

```
IDL> array = FIndGen(25)
IDL> vector = [33.6, 77.2]
IDL> scalar = 5
IDL> text = ['array', 'vector', 'scalar']
IDL> header = 'Test data file.'
IDL> created = 'Created: ' + SysTime()
```

Next, open a file for writing by attaching a logical unit number (which you will have IDL select for you and put into the variable *lun*) to a particular file. Type this:

```
IDL> OpenW, lun, 'test.dat', /Get_Lun
```

Finally, write the data into the file and close the data file. Type this:

```
IDL> PrintF, lun, header
IDL> PrintF, lun, created
IDL> PrintF, lun, array, vector, scalar, text
IDL> Free_Lun, lun
```

Open the file with a text editor and examine it. It will look similar to this:

```
Test data file
Created: Wed May 21 11:43:56 1997
      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
33.6000    77.2000
5
array vector scalar
```

Notice that IDL has put white space between each element of the array variables and that it has started each new variable on its own line. IDL is using an 80 column width by default. If you want a different column width, you can set it with the *Width* keyword to the *OpenW* command.

Reading a Free Format File

Many ASCII files are free format files. For example, files you save from within a spreadsheet program are often free format files. A special kind of free format data is data you read in from the keyboard or from standard input.

In IDL, there are two commands that will let you read free formatted data.

<i>Read</i>	Reads free format data from the standard input or keyboard.
<i>ReadF</i>	Reads free format data from a file.

Rules For Reading Free Format Data

IDL uses the following seven rules to read free format data from either the keyboard or from a file.

1. If reading into a string variable, all characters remaining on the current line are read into the variable.

Look at the first line of your data file. There are three words on the line. What this rule means is that once IDL starts to read into a string variable it doesn't stop until it gets to the end of the line. The reason for this is that of all the data types in IDL, string variables may be any size and, also, blank characters are perfectly good ASCII characters.

Suppose you open the data file for reading and try just to read a single word like this:

```
IDL> OpenR, lun, 'test.dat', /Get_Lun
IDL> word = ''
IDL> ReadF, lun, word
IDL> Print, word
```

What you see is that the entire first line is read into the *word* variable.

```
Test data file.
```

(If you wanted to separate the first line into individual words, you could use IDL's string processing routines to do so.) This ability to read to the end of a line can be a nice feature of IDL. To see why, first rewind the file pointer to point to the beginning of the file. You can use the *Point_Lun* command for this purpose. Type:

```
IDL> Point_Lun, lun, 0
```

Now we can read the first two lines of the data file into a header variable. Type this:

```
IDL> header = StrArr(2)
IDL> ReadF, lun, header
IDL> Print, header
```

These commands read the first two lines of the file and position the file pointer at the start of the array data. You see printed out both header lines on the same output line, like this:

```
Test data file. Created: Wed May 21 14:36:13 1997
```

2. Input data must be separated by commas or whitespace (spaces or tabs).

This is exactly the kind of data that now faces you in the *test.dat* data file. IDL has put five spaces between each element of the *array* variable.

3. Input is performed on scalar variables. Arrays and structures are treated as collections of scalar variables.

What this means is that if the variable you are reading into contains, say, 10 elements, then IDL will try to read 10 separate values from the data file. It will use the next two rules to determine where those values are located in the file.

4. If the current input line is empty, and there are still variables left requiring input, read another line.
5. If the current input line is not empty, but there are no variables left requiring input, ignore the remainder of the line.

To see what this means, try this:

```
IDL> data = FltArr(8)
IDL> ReadF, lun, data
IDL> Print, data
```

You see this:

0.00000	1.00000	2.00000	3.00000	4.00000	5.00000
6.00000	7.00000				

In this case, IDL reads eight separate data values from the file. When it got to the end of the first row of data, it went to the second row (rule 4), because more data remained to be read. You read into the middle of the second row. Now rule 5 comes into play. If you read more data now, you will start on the third data line, because the rest of the second line will be ignored. Try it. Type this:

```
IDL> vector3 = FltArr(3)
IDL> ReadF, lun, vector3
IDL> Print, vector3
```

You see this:

12.0000	13.0000	14.0000
---------	---------	---------

The variable *vector3* contains the values 12.0, 13.0, and 14.0. And now the file pointer is located on the fourth data line in the file (rule 5 again).

6. Make every effort to convert data into the data type expected by the variable.

To see what this means, read the fourth and fifth data lines into a string array, so that the file pointer is positioned at the sixth data line (the line that starts with the value 33.6000). Type this:

```
IDL> dummy = StrArr(2)
IDL> ReadF, lun, dummy
```

Now, suppose you want to read two integer values. IDL makes every effort to convert the data (floating point values in this case) into integers. Type this:

```
IDL> ints = IntArr(2)
IDL> ReadF, lun, ints
IDL> Print, ints
```

You see this:

33	77
----	----

Notice that the floating point values have simply been truncated. There is no attempt to round values to the nearest integer in this conversion process.

7. Complex data must have a real part and an imaginary part separated by a comma and enclosed in parentheses. If only a single value is provided, it is considered the real value and the imaginary value is set to 0. For example, you can read complex data from the keyboard by typing this:

```
IDL> value = ComplexArr(2)
IDL> Read, value
: (3,4)
: (4.4,25.5)
IDL> Print, value
```

Be sure to close the *test.dat* file before you leave this section. Type this:

```
IDL> Free_Lun, lun
```

Examples of Reading and Writing Free Format Files

The easiest way to learn how to read and write data in IDL is to see some examples. Here are a few that illustrate some of the common IDL techniques for reading headers, working with columnar data, and processing the data once it is read into IDL.

Reading a Simple Data File

Start by reading all the data in the *test.dat* file that you just created. First, create the variables that you will be reading from the file. Type this:

```
IDL> header = StrArr(2)      ; Two header lines.
IDL> data = FltArr(5,5)       ; Floating point array
IDL> vector = IntArr(2)       ; Two-element integer vector.
IDL> scalar = 0.0            ; Floating-point scalar.
IDL> string_var = ''         ; A string variable.
```

Notice that the *data* variable is going to be a 5-by-5 array in this case. It was put into the file as a 25-element vector. Reading the data this way is equivalent to reading a 25-element vector and reformatting that vector into a 5-by-5 array. Remember that data in IDL is stored in row order.

Notice, too, that you can't read the text at the end of the file into a three-element string array (which is what you wrote into the data file in the first place) because of rule 1, above. You will have to read this into a scalar string variable and then parse the text strings out of the variable later with IDL string processing commands.

You can read the data from the file all at once. Type this:

```
IDL> OpenR, lun, 'test.dat', /Get_Lun
IDL> ReadF, lun, header, data, vector, scalar, string_var
IDL> Free_Lun, lun
```

To convert the string variable containing the end-of-file text to a three-element string array, you start by trimming the blank characters from both ends of the scalar variable. Type this:

```
IDL> thisString = StrTrim(string_var, 2)
```

Sometimes string processing is easier if you turn strings into byte arrays first. Then you can process the byte arrays, and convert back to strings at the end. This may be a good approach here, although other methods are possible. To convert the string to a byte array, type this:

```
IDL> thisArray = Byte(thisString)
IDL> Help, thisArray
```

You see this is a 19-element byte array. You need to know the ASCII character for a blank space. Use IDL to tell you by typing this:

```
IDL> blank = Byte(" ")
IDL> Print, blank
IDL> Help, blank
```



Note that the return value from the *Byte* command operation on a string is *always* an array. In this case an array of one element. So you are not confused later on, turn this into a scalar value, like this:

```
IDL> blank = blank[0]
```

You see the blank character has an ASCII value of 32.

Use the *Where* command to reveal the blank characters in the byte array, like this:

```
IDL> vals = Where(thisArray EQ blank)
```

Finally, you are ready to turn your string into a three-element string array, like this:

```
s = StrArr(3)
s[0] = String(thisArray[0:vals[0]-1])
s[1] = String(thisArray[vals[0]+1:vals[1]-1])
s[2] = String(thisArray[vals[1]+1:*)])
```

Writing a Column-Format Data File

It is not uncommon to see data stored in files in columns. It is a good idea to know how to read and write this sort of data in IDL, but IDL has a surprise for the unwary programmer. To see what this is, start by writing a column-format data file. Load data into IDL for writing by typing this command:

```
IDL> data = LoadData(15)
```

This data set is a structure that has three fields: *lat*, *lon*, and *temp*, each a 41-element floating point vector. Extract the vectors from the structure by typing these commands:

```
IDL> lat = data.lat
IDL> lon = data.lon
IDL> temp = data.temp
```

Next, open a data file for writing, like this:

```
IDL> OpenW, lun, 'column.dat', /Get_Lun
```

You want to write three columns of data into this file. Using free format output, this can be done in a loop, like this:

```
IDL> PrintF, lun, 'Column data: LAT, LON, TEMP'
IDL> FOR j=0,40 DO PrintF, lun, lat[j], lon[j], temp[j]
IDL> Free_Lun, lun
```

The first four lines of your *column.dat* file should look something like this:

```
Column data: LAT, LON, TEMP
 33.9840      -86.9405      36.9465
 26.2072      -121.615      20.1868
 42.1539      -103.733      231.604
```

Reading a Column-Format Data File

So far, so good. The problem comes when you try to read this column-format data into IDL. You might—quite naturally—try to do it like this. First, create the variables that you will read the data into. Type:

```
IDL> header = ''
IDL> thisLat = FltArr(41)
IDL> thisLon = FltArr(41)
```

```
IDL> thisTemp = FltArr(41)
```

Open the *column.dat* file for reading and read the header line, like this:

```
IDL> OpenR, lun, 'column.dat', /Get_Lun  
IDL> ReadF, lun, header
```

Now, since you put the data into the file using a loop, you might try to read the data out of the file using a loop, too. In other words, you might try to type this:

```
IDL> FOR j=0,40 DO ReadF, lun, thisLat[j], thisLon[j], $  
      thisTemp[j]
```

Unfortunately, this does not work. Although no error is generated by the command above, no data is written into the variables either. (If you print the values of the variables you will see they are all zeros.)

The reason this doesn't work is that there is a hard and fast rule in IDL that states that you *cannot read into a subscripted variable*. The reason for this is that IDL passes subscripted variables into IDL procedures like *ReadF* by *value* and not by *reference*. Data that is passed by value cannot be changed inside of the called routine, since the routine has a *copy* of the data and not a pointer to the data itself. Changing this behavior would require a major re-write of IDL and is not likely to happen.

There are two ways to solve this problem. The first involves reading the data into temporary variables in a loop. This solution is best implemented by using a text editor to enter the commands into a file, since it is difficult to write multiple line loops at the IDL command line. Type this *Point_Lun* command to position the file pointer back at the start of the data file:

```
IDL> Point_Lun, lun, 0
```

Type the following commands in a text file named *loopread.pro*. The file is among those you downloaded to use with this book. Type:

```
temp1 = 0.0  
temp2 = 0.0  
temp3 = 0.0  
ReadF, lun, header  
FOR j=0,40 DO BEGIN  
    ReadF, lun, temp1, temp2, temp3  
    thisLat[j] = temp1  
    thisLon[j] = temp2  
    thisTemp[j] = temp3  
ENDFOR  
END
```

Execute the code in the text file by typing this:

```
IDL> .Run loopread
```

You can see that this works by printing the values of the original vectors and the vectors you just read and seeing that they are the same. For example, type this:

```
IDL> Print, lat, thisLat
```

While this solution works, it may not be the best solution, primarily because it involves a loop, which is inherently slow in IDL. With 41 iterations speed is essentially irrelevant. But if there were 41,000 iterations, execution speed might matter.

If it does, then a better solution is to read the data all at once into a 3 by 41 array, and then pull the vectors out of this larger array using the array processing commands

provided by IDL. To see how this works, rewind the data file again by typing this command:

```
IDL> Point_Lun, lun, 0
```

Next, read the data all at once into a 3 by 41 floating point array, like this:

```
IDL> header = ''
IDL> array = FltArr(3, 41)
IDL> ReadF, lun, header, array
```

Parse the vectors out of this large array using array subscripts, like this:

```
IDL> thisLat = array[0,*]
IDL> thisLon = array[1,*]
IDL> thisTemp = array[2,*]
IDL> Free_Lun, lun
```

Notice, however, that these new vectors are column vectors (i.e., they are 1 by 41 two-dimensional arrays). Type this:

```
IDL> Help, thisLat, thisLon, thisTemp
```

To make these column vectors into the more familiar row vectors, the *Reform* command is used to reform the 1 by 41 arrays into 41 by 1 arrays. When the last dimension of a multi-dimensional array is one, IDL drops that dimension. Type this:

```
IDL> thisLat = Reform(thisLat)
IDL> thisLon = Reform(thisLon)
IDL> thisTemp = Reform(thisTemp)
IDL> Help, thisLat, thisLon, thisTemp
```

Creating a Template for Reading Column-Format Data

Since many people have column-format data files, IDL 5 introduced new routines to make it easier to read this type of data file. The help is in the form of two new commands: *ASCII_Template* and *Read_ASCII*. The *ASCII_Template* command is a widget program that takes you through the steps necessary to define your column data. You can give each column of data a name, tell IDL what type of data it is, and even skip data columns if you like. The result of running the program is a “template” of the data file in the form of an IDL structure variable. This template can be passed to the *Read_ASCII* command and the data will be read according to the template specifications. The result is an IDL structure variable with fields named and typed according to your specifications. To see how it can be used with the file above, type:

```
IDL> fileTemplate = ASCII_Template('column.dat')
```

Follow the directions on the widget dialog form that appears on the display. There are three “pages” to the form. On the first page you see a representative sample of lines from the data file, with their line numbers on the left. In the text widget labeled *Data Starts at Line*: type in the number 2. This will allow the one line header to be skipped in this file. Click the *Next* button in the bottom right corner of the widget to move to the next page.

Notice on this page that the number of fields per line is listed as three and that the *White Space* button is selected for the data delimiter. This information is correct, so just hit the *Next* button to get to the final page.

This is the page where the columns in the data set can be named and their data type specified. Notice that if you want to skip one or more of the columns in the data set that the *Type* dropdown in the upper right-hand corner of the display can be set to a type of *Skip Field*. Name the three fields *Latitude*, *Longitude*, and *Temperature*,

respectively, by typing in the *Name* text widget at the upper right of the form. All three fields are floating point type.

When you are finished naming and typing each column of data, click the *Finish* button on the form. The result is an IDL structure variable that describes the data in the file and can be used as input into the *Read_ASCII* command. Type:

```
IDL> Help, fileTemplate, /Structure
```

To read the data in the file, use the *Read_ASCII* command like this:

```
IDL> data = Read_ASCII('column.dat', Template=fileTemplate)
```

The data is read immediately and the result is an IDL structure variable containing three fields, labeled *latitude*, *longitude*, and *temperature*. Type:

```
IDL> Help, data, /Structure
```

If you want to pull the vectors out of the structure, you can type this:

```
IDL> thisLat = data.latitude  
IDL> thisLon = data.longitude  
IDL> thisTemp = data.temperature
```

The *ASCII_Template* and *Read_ASCII* commands can be used with either free formatted data files or with the explicitly formatted data files described below.

Writing with an Explicit File Format

To read and write an explicit file format, use the same *ReadF* and *PrintF* commands that were used for free format files, except now the file format is explicitly stated with a *Format* keyword. (The *Format* keyword can also be used with the *Read* and *Print* commands when reading and writing to standard input and output.)

The syntax of the *Format* keyword is similar to the format specifications you may have used in a FORTRAN program. Although the format specifications can be quite complex, here are a few of the most common. You will immediately recognize them if you have written any FORTRAN code to read or write data.

A Few Common Format Specifiers

There are many format specifiers in IDL. Here are a few of the most common ones. Consider the vector *data*, defined like this:

IDL> data = FIndGen(20)	
I	Specifies integer data. Print the vector out as two-digit integers, five numbers on each line, each number separated by two spaces:
IDL> thisFormat = '(5(I2, 2x),/)' IDL> Print, data, Format=thisFormat	
F	Specifies floating point data. Write the data out as floating values with two digits to the right of the decimal place, one value per line. (Be sure to include enough space in the number for the decimal point itself.)
IDL> thisFormat = '(F5.2)' IDL> Print, data, Format=thisFormat	
D	Specifies double precision data. Write out five numbers per line, four spaces between each number, in double precision values with 10 digits to the right of the decimal point.
IDL> thisFormat = '(5(D13.10, 4x))'	

```

IDL> Print, data*3.2959382, Format=thisFormat
E           Specifies floating point data using scientific notation (e.g.,
           116.36E4).

IDL> thisFormat = '(E10.3)'
IDL> Print, data*10E3, Format=thisFormat

A           Specifies character data. Convert the number to strings and
           write them out as four-character strings, each separated by 2
           blank spaces, with four numbers to a line:

IDL> thisFormat = '(4(A4, 2x))'
IDL> Print, StrTrim(data,2), Format=thisFormat

nX          Skip n character spaces.

```

Writing a Comma Separated Explicitly Formatted Data File

Sometimes data files must be written with an explicit format so they can be read by other software. A comma separated data file is a common example of such a file. For example, you may want the data vectors you read above to be written to such a file. Here is how to do it. Open a file named *format.dat* for writing, like this:

```
IDL> OpenW, lun, 'format.dat', /Get_Lun
```

Create a string variable that is the comma, like this:

```
IDL> comma = ','
```

Now, write the file out, using an explicit file format of a floating value 10 digits wide with three digits to the right of the decimal, followed by a comma and two spaces, like this:

```

IDL> thisFormat = '(F10.3, A1, 2x, F10.3, A1, 2x, F10.3)'
IDL> FOR j=0,40 DO PrintF, lun, thisLat[j], comma, $
      thisLon[j], comma, thisTemp[j], Format=thisFormat
IDL> Free_Lun, lun

```

The first three lines of the data file looks like this:

```

48.000,      -121.128,      36.946
44.843,      -108.133,      163.027
29.865,      -109.668,      89.870

```

Reading a Comma Separated Explicitly Formatted Data File

To read the data file you just created as an explicitly formatted data file, type this:

```

IDL> OpenR, lun, 'format.dat', /Get_Lun
IDL> thisFormat = '(2(F10.3, 3x), F10.3)'
IDL> array = FltArr(3, 41)
IDL> ReadF, lun, array, Format=thisFormat
IDL> Free_Lun, lun

```

It is as simple as that. Notice that you skipped over the commas by treating them as spaces in the format specifier above.

Reading Formatted Data From a String

The useful IDL command, *ReadS*, can read free format or explicitly formatted input from a string variable instead of from a file. *ReadS* uses the same rules for reading formatted data that the commands *Read* and *ReadF* use. In other words, using *ReadS* is like reading from a data file, except that the read occurs on a string variable.

This command is especially useful when numerical information is needed from a file header. For example, suppose the first line in an ASCII data file indicates the number of columns and rows in the data file, followed by the date the data was collected, like this:

```
10      24500      12 June 1996
```

This header could be read from the file and a properly sized data array created for reading the data, like this:

```
firstLine = ''  
ReadF, lun, firstLine  
columns = 0  
rows = 0  
date = ''  
ReadS, firstLine, columns, rows, date  
dataArray = FltArr(columns, rows)
```

Reading and Writing Unformatted Data

Sooner or later, data begins to overwhelm either you or your machine. In either case, you begin to think about better ways to store it. Unformatted data (sometimes called binary data) is much more compact than formatted data and is often used with large data files. There are two commands to read and write unformatted data that are the equivalent of the *ReadF* and *PrintF* commands used earlier to read and write formatted data files. These are the *ReadU* and *WriteU* commands.

Unformatted data files are basically stored as one long stream of bytes in a file. What those bytes mean (i.e., how those bytes are translated into data of a particular data type and organization) is hard to figure out unless you know what was put into the file in the first place. Inside a file, one byte looks pretty much like any other. Sense is made of the bytes by reading the bytes into the properly typed and structurally organized variables. In principle, this is easy to do, since most data types have a defined byte size. Floating point values, for example, are each four bytes in size. IDL integers are each two bytes, and so on.

To read unformatted data files, simply define your variables, open the file for reading, and read the bytes into those variables—one after the other—with the *ReadU* command. Each variable reads as many bytes out of the file as it requires, given its data type and organizational structure. A five-element floating point vector, for example, will read five (number of elements) times four (number of bytes in a floating point value), for a total of twenty bytes, from the file.

Reading an Unformatted Image Data File

For example, suppose you want to read one of the several unformatted image data files located in the IDL *examples/data* subdirectory. These files happen to contain byte data, but they could just as easily contain integer or floating point data. Here are commands that can be used to open one of these files, the *galaxy.dat* file.

```
IDL> filename = Filepath(Root_Dir=!Dir, $  
SubDirectory=['examples','data'], 'galaxy.dat')  
IDL> OpenR, lun, filename, /Get_Lun
```

This particular image is organized as a 256 by 256 array. If you didn't know this ahead of time, you would probably have a hard time reading the data properly, since there is nothing in the data file that indicates how the bytes should be organized. (Which is, by the way, an excellent reason to learn about HDF files in the next chapter. HDF files

contain information about the data inside the file. This information, called *metadata* or *attributes*, can be used to read the data properly.)

About the only thing you can learn about an unformatted data file without prior knowledge is how big it is. That is, how many bytes it contains. For example, the *FStat* (*File Status*) command can tell you the size of this file in bytes, like this:

```
IDL> fileInfo = FStat(lun)
IDL> Print, fileInfo.size
65536
```

There are 65,536 bytes in this file, but this gives you no clue as to how the data is organized structurally. For example, the bytes could be arranged as a 128 by 512 two-dimensional array or as a 64 by 64 by 16 three-dimensional array. There is no way to know. Desperate programmers sometimes try various combinations of arrays sizes and then display the data. If it doesn't "look" right, they try another size, and so on.

In this case, the data should be arranged as a 256 by 256 byte array, so the *image* variable can be set up like this:

```
IDL> image = ByteArr(256, 256)
```

Now, read the data from the file and close the file, like this:

```
IDL> ReadU, lun, image
IDL> Free_Lun, lun
```

To display the data, type this:

```
IDL> Window, XSize=256, YSize=256
IDL> Device, Decomposed=0
IDL> TVScl, image
```

Writing an Unformatted Image Data File

Suppose you performed some image processing on this image and you wanted to save the results of the processing in another data file. For example, you could perform a Sobel edge enhancement operation on the image, like this:

```
IDL> edge = Sobel(image)
```

The Sobel operation has the effect of not only providing an image that has its edges enhanced, but the returned image is no longer of byte type. In fact, it is integer data, as you can see by typing this:

```
IDL> Help, image, edge
```

Integers are two-byte values in IDL, so if this data is written into a data file, the file will be twice the size of the original byte data file. This might be confusing to the person who is trying to read the processed image data file, so you might want to put some information into the file to give the user some help about the kind of data in the file and how the data should be organized. This file information might be defined like this:

```
IDL> fileInfo = 'Sobel Edge Enhanced, 256 by 256 INTEGERS'
```

This string will be written into the file in front of the image data.

But wait. The string *fileInfo* is just a string of bytes, too. In this case, it is a string 31 bytes long. If you don't know this about the unformatted file, then it will be very difficult to read the data out of the file later on. Imagine, for example, that you thought the file information string was 30 bytes long. Every integer (two bytes) that you read from the file after reading the file information string will be the wrong value completely!

To get around this significant limitation, most headers in unformatted data files have a defined size (usually some multiple of 256). Suppose, for example, that you decide that all image files will have a 512-byte header. You could use the *Replicate* and *String* commands in IDL to create a string filled with blank characters that was exactly 512 bytes long, like this:

```
IDL> header = String(Replicate(32B, 512))
```

The byte value 32 is the ASCII value of a blank character. To insert the file information string into this longer header string to create a header of the proper size, you can use the *StrPut* (Put a String inside of another string) command in IDL, like this:

```
IDL> StrPut, header, fileInfo, 0
```

Finally, the header and the data can be written into a new unformatted data file, like this:

```
IDL> OpenW, lun, 'process.dat'  
IDL> WriteU, lun, header, edge  
IDL> Free_Lun, lun
```

When strings are written to an unformatted file, just exactly the number of bytes that comprise the string are written into the file.

Reading Unformatted Data Files with Headers

Suppose you want to read the data file you just created above. There are 512 bytes of header information, followed by $256 \times 256 \times 2$ bytes of image data. You would like the header information as a string, of course, because it contains textual information about what kind of data is in the file.

With formatted data, header variables can be created either as null strings or as an array of null strings and then entire lines of a data file can be read into the header variable all at once. This is not possible with unformatted files. In fact, the rule on unformatted string data is that when reading strings out of a file, just exactly enough bytes are read to fill the current length of the string. Thus, a header defined as a null string would read *no* bytes out of an unformatted file!

This means that you must know the length of string you are reading before you try to read it out of the file. In the case of the *process.dat* file you just created, the header is 512 bytes long. You could use the *String* and *Replicate* commands to create a blank string of the proper length to read into, like before, but it's often easier to read the header into a byte array variable and then convert that to a string later. The code might look like this:

```
IDL> OpenR, lun, 'process.dat', /Get_Lun  
IDL> header = BytArr(512)  
IDL> ReadU, lun, header  
IDL> Print, String(header)
```

Sobel Edge Enhanced, 256 by 256 INTEGERS

From this information the proper data array can be created and the image data read from the file and displayed, like this:

```
IDL> edgeImage = IntArr(256, 256)  
IDL> ReadU, lun, edgeImage  
IDL> Free_Lun, lun  
IDL> Window, XSize=256, YSize=256  
IDL> TV, edgeImage
```

Problems with Unformatted Data Files

Unfortunately, as nice as unformatted data is to work with, there are also problems associated with using it. For one thing, unformatted data is extremely machine specific. Data written on a Sun computer cannot always be read on an SGI or HP computer and certainly not on a PC or Macintosh computer, without considerable mucking around. (The *ByteOrder* command can be used to address many of these problems and the new *Swap_If_Big_Endian* and *Swap_If_Little_Endian* keywords introduced in IDL 5.1 for the *Open* commands are absolutely essential when writing code to read binary data on a variety of machine architectures.)

To make it possible to transport unformatted data across machine architectures, IDL supports XDR, or eXternal Data Representation, file format. The XDR format is a public domain data format created by Sun Microsystems. It is available on almost all modern computers. It stores a small amount of metadata (extra information about the data itself) in a binary file. But XDR files are still compact.

If files are written in the XDR unformatted format, then data files can be transferred between computers easily. In other words, XDR unformatted files become portable across machine architectures.

To read or write a file in the XDR format, the file must be opened with the *XDR* keyword set. For example, to write the *process.dat* file above as an XDR file you would type this:

```
IDL> OpenW, lun, 'process.dat', /Get_Lun, /XDR
```

The normal *WriteU* command is used to write data to the file, like this:

```
IDL> WriteU, lun, header, edge
IDL> Free_Lun, lun
```

The length of strings in XDR files is stored and restored along with the string itself. This means you don't have to initialize string variables to their correct length like in normal unformatted file. For example, to open and read the information in this XDR file, you can type this:

```
IDL> OpenR, lun, 'process.dat', /XDR
IDL> thisHeader = ''
IDL> thisData = IntArr(256, 256)
IDL> ReadU, lun, thisHeader, thisData
IDL> Free_Lun, lun
```

Accessing Unformatted Data Files with Associated Variables

Large unformatted data files often consist of a series of repeating units. For example, a satellite may take a 512 by 600 floating point image every half hour and store the images one after the other in a single data file that is downloaded at regular intervals. It is not unusual to see data files that contain 50-100 MBytes of data in them. An IDL associated variable is often the best (sometimes only) way to work with the type of data.

An *associated variable* is a variable that maps the organizational structure of an IDL array or structure variable onto the contents of a data file. The file is treated as an array of these repeating units. The first unit has an index of 0, the second an index of 1, and so on. An associated variable does not keep the entire data set in memory like a normal variable. Instead, when an associated variable is referenced, IDL performs the requested input or output on just that portion of the data required, and this is what goes into memory.

Advantages of Associated Variables

There are several advantages to the associated variable method:

1. File input and output occurs when the variable is used in an expression. A separate read or write command is not necessary.
2. The size of the data set is not limited by memory, as it can be sometimes for large data sets. Data sets too large for physical memory can be handled easily by breaking the data into “chunks” of data.
3. There is no need to declare the number of arrays or structures that are mapped onto the data set ahead of time.
4. Associated variables are the most efficient form of I/O.

Defining Associated Variables

To define and use an associated variable, open the data file in the usual way and then use the *Assoc* command to create the associated variable. For example, you can open the file *abnorm.dat* in the *examples/data* subdirectory of the main IDL directory, like this:

```
IDL> filename = Filepath(Root_Dir=!Dir, 'abnorm.dat', $  
SubDirectory=['examples','data'])  
IDL> OpenR, lun, filename, /Get_Lun
```

This file contains 16 images or frames, each of which is a 64 by 64 byte array. Create the associated variable for this data set like this:

```
IDL> image = Assoc(lun, BytArr(64, 64))
```

The first argument to the *Assoc* command is the logical unit number of the file that is associated with the variable *image*. The second argument is a description of the unit that is repeated in the file.

Although it is not the case with this file, these files often have header information in front of the repeating file unit. If that is the case, a third positional argument can be given to the *Assoc* command that specifies the size of this header or offset into the file. For example, suppose that the first 4096 bytes of this *abnorm.dat* file was header information and you wished to skip this header in the file. Then the *Assoc* command could be written like this:

```
IDL> image2 = Assoc(lun, BytArr(64, 64), 4096)
```



Note that you now have two variables, *image* and *image2*, associated with the same data file. This is perfectly legal in IDL and, in fact, can be a good way to access unformatted data that does not have uniformly-sized repeating units in it. By changing the offset into the file it is possible to use the associated variable method as a sort of random-access method of reading and writing data.

To display the fifth frame or image in variable named *image* above, type this:

```
IDL> Device, Decomposed=0  
IDL> TvScl, image(4)
```

The data is read from the data file into a temporary variable, which is displayed and then discarded by IDL. No explicit *ReadU* command is required and no permanent memory has been used in IDL to work with this image. If you wish to make a variable from an associated variable, you create the variable in the usual way. For example, you can type this:

```
IDL> image5 = image(4)  
IDL> TV, Rebin(image5, 256, 256)
```

The type of repeating unit in the data file does not have to be a simple 2D image array as it is here. It can be a complicated structure. For example, each repeating unit might consist of a 128 byte header, two 100-element floating point vectors and a 100 by 100 integer array. If this was the case, an associated variable might be created for the file like this:

```
OpenR, 10, 'example.dat'
info = BytArr(128)
xvector = FltArr(100)
yvector = FltArr(100)
data = IntArr(100, 100)
struct = {header:info, x:xvector, y:yvector, image:data}
repeatingUnit = Assoc(10, struct)
```

Since the variable that is mapped to this data file is a structure variable, you must make a temporary copy of the structure before it can be de-referenced. For example, to display the image portion of the third repeating unit in the file, you would type:

```
tempVariable = repeatingUnit(2)
TvScl, tempVariable.image
```

The connection between an associated variable and a file is closed in the usual way with the *Free_Lun* or *Close* commands, like this:

```
Free_Lun, lun
Close, 10
```

Reading and Writing Files with Popular File Formats

So far, this chapter has talked about the general ways IDL can read and write data files. These low-level capabilities make it possible to read and write many kinds of data files in IDL. But there are many other data file formats you will probably want to know how to read and write, too. Among these are formats like GIF, JPEG, and TIFF, which are often used to share data with colleagues across the office or across the world. You will also want to know how to create PostScript files, which will almost certainly be required if you wish to publish your graphical output in hardcopy.

IDL has the ability to read and write many popular file formats. A list of these formats is shown in Table 10. In this section you will learn how to read and write GIF, JPEG, TIFF, and DICOM medical image files. I choose these formats because if you know how to read and write each of these, which are all slightly different from one another, you will know how to read or write almost any of the formats IDL provides. In the following chapter, I introduce you to HDF format files which, along with CDF and net-CDF file formats, make up what we call the “scientific data file formats”. These formats have the capability of storing “meta-data”, or information about the data itself, in the same file with the data. And then in the chapter after that, you learn how to write PostScript output, either sending the output directly to the printer or by saving the output in PostScript files.

Querying Image Files for Information

In IDL 5.2, RSI made it much easier to gather information about data stored in popular file formats by introducing file “query” commands. These commands allow you to query the data file without having to actually read the data. The commands are able to access the metadata (information about the data) that is stored inside the file along with the image data itself. Here is a list of the new file query commands: *Query_BMP*, *Query_DICOM*, *Query_GIF*, *Query_JPEG*, *Query_PICT*, *Query_PNG*, *Query_PPM*, *Query_SRF*, *Query_TIFF*, and *Query_WAV*.

File Format	IDL Routine to Read	IDL Routine to Write
BMP	Read_BMP	Write_BMP
CDF	See CDF library	See CDF library
DICOM	IDLffDICOM object	None
DXF	IDLffDXF object	IDLffDXF object
GIF (Removed in IDL 5.4)	Read_GIF	Write_GIF
HDF	See HDF library	See HDF library
HDF-EOS	See HDF library	See HDF library
Interfile	Read_Interfile	None
JPEG	Read_JPEG	Write_JPEG
netCDF	See netCDF library	See netCDF library
PICT	Read_PICT	Write_PICT
PBM/PPM	Read_PPM	Write_PPM
PNG	Read_PNG	Write_PNG
PostScript	None	PS or PRINTER device
Sun Rasterfiles	Read_SRF	Write_SRF
SYLK	Read_SYLK	Write_SYLK
TIFF/GeoTIFF	Read_TIFF	Write_TIFF
WAVE	Read_WAVE	Write_WAVE
X11-bitmap	Read_X11_Bitmap	None
XWD	Read_XWD	None

Table 10: *IDL can read and write many popular data file formats, often by means of library routines written in the IDL language or by dynamic link modules (DLM) that are added to IDL at run-time. The CDF, netCDF, and HDF file formats are known collectively as scientific data formats and have their own IDL interface and library of routines. See “Reading and Writing HDF Data” on page 161 for more information about HDF file formats.*

The query commands all work in the same way. They are functions that return a value of 0 or 1 to indicate if they were successful (indicated by a 1) reading the metadata of an image file. If they successfully read a file, an IDL structure variable containing information about the file is returned to the user as an output variable. Users can access the fields of this structure to obtain information about the file.

For example, to query the JPEG file, *rose.dat*, in the IDL *examples/data* subdirectory and return information about the file in the variable *fileinfo*, type this:

```
IDL> filename = Filepath('rose.jpg', $
SubDir=['examples', 'data'])
IDL> ok = Query_JPEG(filename, fileinfo)
```

To see what kind of information was returned, type:

```
IDL> Help, fileinfo, /Structure
```

You see printed out the following information:

** Structure <1364998>, 7 tags, length=36, refs=1:
CHANNELS LONG 3
DIMENSIONS LONG Array [2]
HAS_PALETTE INT 0
IMAGE_INDEX LONG 0
NUM_IMAGES LONG 1
PIXEL_TYPE INT 1
TYPE STRING 'JPEG'

You can see that there is one image in this file (*Num_Image*=1), that it is byte type data (*Pixel_Type*=1), and that it is a 24-bit image (*Channels*=3). The size of the image can be seen by printing out the dimensions field, like this:

```
IDL> Print, fileinfo.dimensions
227    149
```

Other image query routines will have similar fields in their return structures.

Creating a Graphic Display Program

To begin learning how to read and write files in popular file formats, let's first write a program to display a graphic. Open a new text editor file and type the following commands. If you prefer not to type his file, it is already written for you and among the programs you downloaded to use with this book. It is named *columnavg.pro*. The purpose of the program is to display an image, with the average value of each column of image data plotted above it. You will learn how to write a graphic display program later (see "Writing an IDL Graphics Display Program" on page 239, for example). For now, just observe that this program uses techniques that have already been explored in earlier chapters of this book.

```
PRO ColumnAvg, image
On_Error, 2
; Make sure an image is available.
IF N_Elements(image) EQ 0 THEN image = LoadData(7)
; Find the size of the image.
ndims = Size(image, /N_Dimensions)
IF ndims NE 2 THEN Message, 'Image parameter must be 2D.'
s = Size(image, /Dimensions)
xsize = s[0]
ysize = s[1]
; Load colors for graphic.
backColor = GetColor('gray', !D.Table_Size-2)
axisColor = GetColor('navy', !D.Table_Size-3)
dataColor = GetColor('green', !D.Table_Size-4)
LoadCT, 33, NColors=!D.Table_Size-4
; Save system variables.
theFont = !P.Font
thePlots = !P.Multi
; Set system variables and create the data.
```

```
!P.Font = 0
!P.Multi=[0,1,2]
columnData = Total(image,2)/ysize

; Draw the plots.

Plot, columnData, /NoData, Color=axisColor, $
    Background=backColor, XStyle=1, XRange=[0,xsize-1], $
    Title='Column Average Value', XTitle='Column Number'
OPlot, columnData, Color=dataColor
TVImage, BytScl(image, Top=!D.Table_Size-5)

; Restore system variables.

!P.Multi = thePlots
!P.Font = theFont
END
```

Save the file as *columnavg.pro*, compile it, and run it like this:

```
IDL> .Compile columnavg
IDL> Window, Title='Column Average Plot', XSize=400, $
    YSize=400
IDL> ColumnAvg
```

The output should look like the illustration in Figure 70, below.

Creating Color GIF Files

GIF files are often used to publish graphical information on the World Wide Web. If you want to share your graphical results with colleagues, sooner or later you will want to read or write a GIF file.



Note that starting in IDL 5.4 you must have a letter from the GIF file format patent holder (Compuserve) granting you permission to create GIF files. Research Systems will supply you with the proper license for GIF file formation upon receipt of such a letter. Contact Research Systems technical support for complete information.

GIF files are written in IDL with the *Write_GIF* command, which has the form:

```
Write_Gif, filename, image2d, r, g, b
```

Where the variable *filename* is the name of the output file, *image2d* is a 2D byte image array, and the variables *r*, *g*, and *b* are the color table vectors that are required to be present with the 2D image array if the image is to be displayed in color.

The filename is often obtained from the user by means of the *Dialog_Pickfile* command, like this:

```
IDL> filename = Dialog_Pickfile(/Write, $
    File='columnavg.gif', Filter='*.gif', $
    Title='Select GIF File for Writing...')
```

Notice that a default file name is given. This is done in a quixotic attempt to keep the user from having to type the name. Perhaps they can just click the *Accept* button. (Research has shown that a typical computer user will make at least four typing errors when typing something as long as a file name. It is good to keep this in mind when you are writing your IDL programs. Ask a user to select a file name with the mouse, rather than type it, whenever possible.)

The next step is to obtain the 2D image array (as byte values) and the color table vectors that go with the image. How you do this depends upon what kind of display device you are on, and in particular, it depends upon the depth (how many bits your

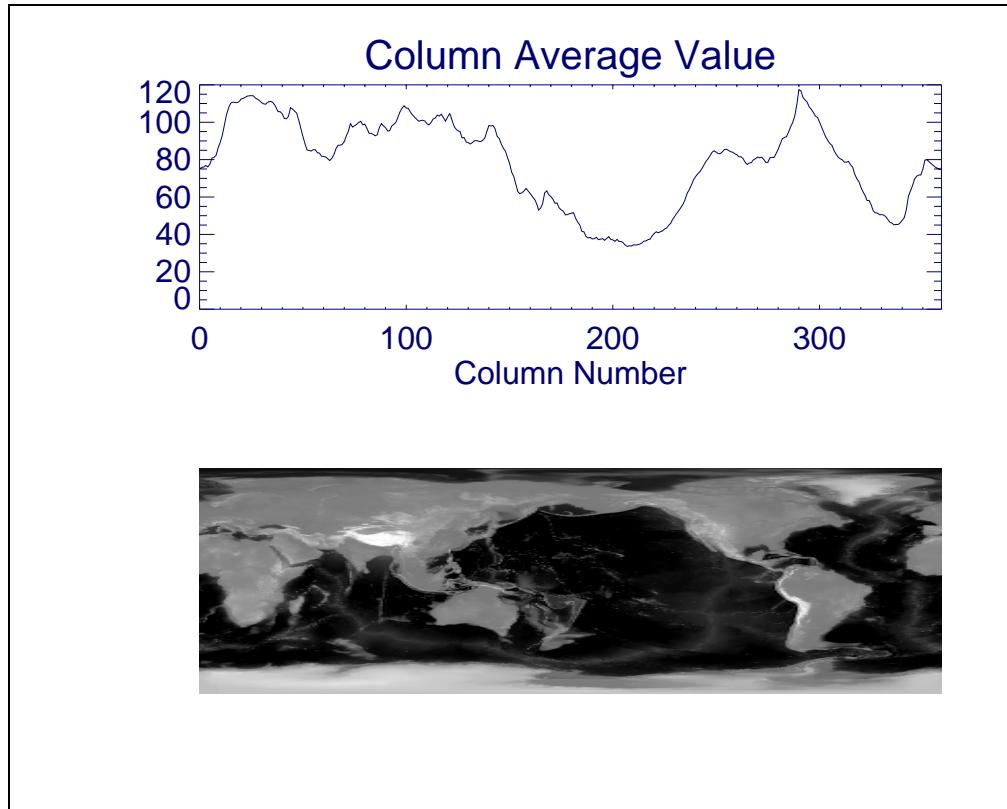


Figure 70: The *ColumnAvg* program that will be saved as a GIF, JPEG, and TIFF file.

display device supports) of your display device. To see the depth of your current display device, type this:

```
Device, Get_Visual_Depth=thisDepth & Print, thisDepth
```

If the depth of your display device is eight, you will obtain the image and color table vectors in one way, if the display device depth is greater than eight, you will obtain them another way. If you are writing platform-independent IDL code, you will have to check the depth of the display device and execute the appropriate sequence of commands depending on the depth of the device.

If the Display Depth is Eight

If you are on an 8-bit display device, your job is fairly simple. You can obtain the current color table vectors by running the *ColorAvg* program and then using the *TVLCT* command with the *Get* keyword, like this:

```
IDL> TVLCT, r, g, /Get
```

The color vectors must be 256 elements long. These are probably not that long if you are running IDL on an 8-bit display, but don't worry. If they are not long enough when you write the GIF file, IDL will pad them with zeros for you. If you want to take full advantage of all 256 colors, consider loading your color table and obtaining the color vectors in the Z-graphics buffer, which has 256 colors available by default. See “Graphics Display Tricks in the Z-Graphics Buffer” on page 120 for additional information. Another alternative is to obtain the color table vectors from an *IDLgrPalette* object.

To obtain the 2D byte array, all you need is to get a screen dump of the current graphics window. That is done with the *TVRD* command, like this:

```
IDL> image2d = TVRD()
```



Note that if you already have a 2D byte array, there is no need to copy the graphics window. We copy it here because we want the GIF file to contain the entire graphic display, not just the image data.

If the Display Depth is Greater than Eight

If the display depth is greater than eight, then you have just a bit more trouble ahead of you in creating a GIF file. Because, remember, when you take a screen dump on a 16-bit or 24-bit display you obtain a 24-bit image with the color information built into the image itself. (See “Obtaining Screen Dumps on 24-Bit Displays” on page 73 for additional information.) In other words, the *TVRD* command should be used like this:

```
IDL> Device, Decomposed=1
IDL> image24 = TVRD(True=1)
```

What you have to do is go from a 24-bit image with colors built into the image itself to a 2D image and the appropriate color table. You see an illustration of the problem in Figure 71.

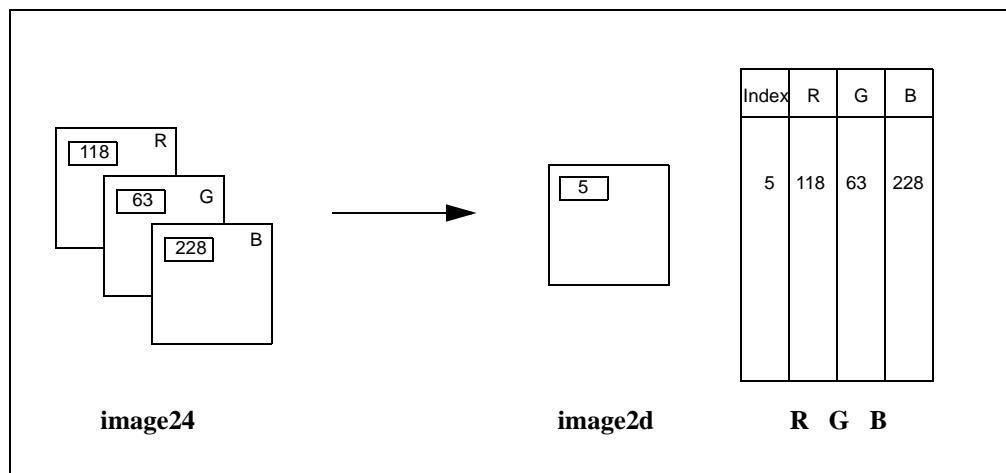


Figure 71: To create a GIF file on a 24-bit display, you have to take a 24-bit image screen dump and extract a 2D image array and the appropriate color vectors. This is done with the *Color_Quan* command.

This is done with the *Color_Quan* command in IDL. With *Color_Quan* you can select either of two different algorithms for extracting the color information from the 24-bit image: a statistical method that attempts to find the N best colors that represent the original colors in the image, or a method that divides the color space into a cube and uses a dithering method to select colors. I've found the statistical method (which I illustrate here) the best when there are many colors in the graphic display, and the color cube method (which is selected by means of values to the *Cube* keyword) is better if you have just a few colors in your graphic or your graphic is rendered in shades of gray. Be sure to read the *Color_Quan* documentation if you are not happy with the result. And please understand that the results of *Color_Quan* are seldom as good as the original 24-bit display. The command looks like this:

```
IDL> image2d = Color_Quan(image24, 1, r, g, b)
```

The number 1 as the second parameter indicates the pixel interleaving in the 24-bit image. It should be the same value as you used for the *True* keyword in the *TVRD*

command above. The variables *r*, *g*, and *b* are output variables that now contain the color table vectors.

Writing the GIF File

The final step, in either case, is to write the GIF file, like this:

```
IDL> Write_GIF, filename, image2d, r, g, b
```

That's all that is necessary. You don't need to obtain a logical unit number or anything else. All of these details are handled by the *Write_GIF* command.

If you have an application that can open and read a GIF file, try to read this one you just created. Many World Wide Web browsers support reading GIF files. See if your browser has an *Open File* button with which you can try to read this GIF file.

Reading a GIF File

To read the GIF file you just created in IDL, simply use the *Read_GIF* command to read the image and color vectors out of the GIF file, like this:

```
IDL> Read_GIF, filename, thisImage, rr, gg, bb
```

The variables *thisImage*, *rr*, *gg*, and *bb* are all output variables that are filled by the *Read_GIF* program.

Clear the graphics window and load a gray-scale color table, so you can see what happens next as you display this image. Type this:

```
IDL> Erase
```

Now, display the image you just read out of the GIF file, like this:

```
IDL> Device, Decomposed=0
IDL> TV, thisImage
```

Chances are what you see in the graphics window is not what you expected. The colors will probably look strange. This is because the color table the GIF image uses is *not* like the color table you had loaded previously. To make the image appear in its correct colors you must load the color table associated with the GIF image. Type this:

```
IDL> TVLCT, rr, gg, bb
IDL> TV, thisImage
```

You should now see the original image in the display window.

But now have a look at the colors you have loaded in your color table. Use the program *CIndex*, which is one of the programs you downloaded to use with this book, to view the current color table colors:

```
IDL> CIndex
```

This color table doesn't look anything like the ones normally used in IDL. In fact, it is more like the color tables used in the IDL object graphics system. And there are consequences of that. For example, try using the *TVImage* command, like this:

```
IDL> Window, XSize=500, YSize=450
IDL> TVImage, thisImage
```

Whoops! What has happened here!?

What has happened is that *TVImage* interpolates image values as it resizes images to fit into a window. So, for example, if adjacent pixels have values of 5 and 7 and a third pixel is added between them, the pixel will have a value of 6. In most color tables, the colors at indices 5, 6, and 7 will be pretty much the same. But that is *not* the case here.

In fact the color at index 6 has no relationship at all to the colors at 5 and 7; it is just a color.



So GIF files that come with color tables cannot be resized with bilinear interpolation. They must be resized by replicating the pixel values that are already in the image. For example, this image should be resized by using the *NoInterpolation* keyword to *TVImage*, like this:

```
IDL> TVImage, thisImage, /NoInterpolation
```

Creating Color JPEG Files

Another file format that is often used to share graphical results on the World Wide Web is the JPEG format. The JPEG format is called a *lossy* compression format. That is, the image data is compressed when it goes into the file and some of the information content of the data is lost and cannot be recovered. The amount of compression, and thus the amount of information that is lost and the quality of the output image, can be set using a quality index value. Values range from 0 (poor quality with much information content lost) to 100 (excellent quality with little or no information content lost). Normally, the quality index is set to about 75, which provides an adequate amount of compression without a noticeable loss of information content and image quality.

A color JPEG image will be a 24-bit image. That is, the image will be a 3D byte array, in which one of the dimensions will be a 3. The placement of this dimension will determine if the image is pixel-interleaved (3, m, n), row-interleaved (m, 3, n), or band-interleaved (m, n, 3). The JPEG file is written with the *Write_JPEG* command in IDL. The general form of the command looks like this:

```
Write_JPEG, filename, image24, Quality=75, True=1
```

The variable *filename* is the name of the output file, the variable *image24* is a 24-bit image with the color information built in, the *Quality* keyword is a value from 0 to 100 that selects the amount of image compression (and, hence, image quality), and the *True* keyword selects the type of pixel interleaving.

As with the GIF file above, how you create the JPEG file depends upon the screen depth of the machine running IDL. To set the stage, type these commands:

```
IDL> Window, Title='Column Average Plot', XSize=400, $  
      YSize=400  
IDL> ColumnAvg
```

First, you obtain a file name for the output file:

```
IDL> filename = Dialog_Pickfile(/Write, $  
      File='columnavg.jpg', Filter='*.jpg', $  
      Title='Select JPEG File for Writing...')
```

Next, you find the depth of the current display device, by tying this:

```
IDL> Device, Get_Visual_Depth=thisDepth & Print, thisDepth
```

What you do next depends on this value. And, in fact, the problem is just the opposite of the problem we just dealt with in creating a GIF file. Here life is easy if we are on a 24-bit display, but more difficult if we are on an 8-bit display.

If the Display Depth is Eight

If you are on an 8-bit display device, you obtain the 2D image array and the color table vectors like this:

```
IDL> TVLCT, r, g, b, /Get  
IDL> image2d = TVRD()
```

But now you have the task of creating a 24-bit image out of this 2D image array and the color table vectors. The problem is shown in Figure 72, below.

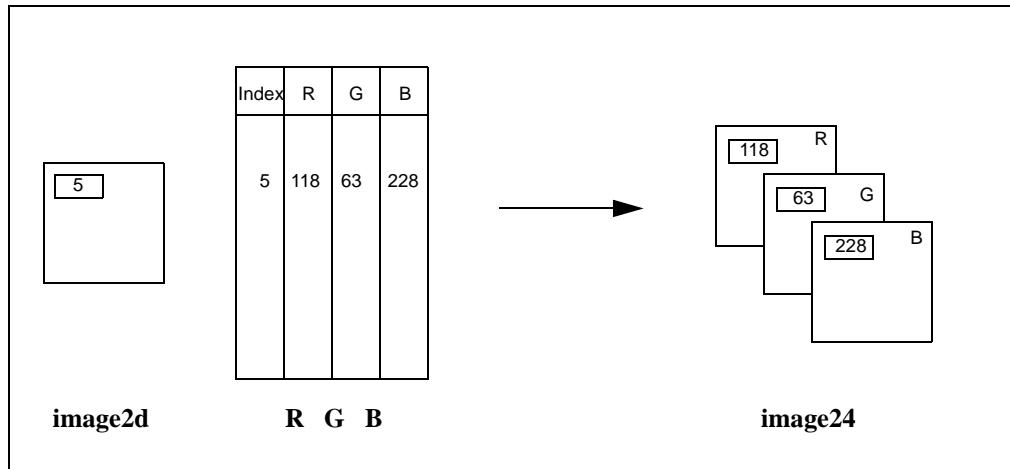


Figure 72: With JPEG files you often have the opposite situation from a GIF file. Here you have to construct a 24-bit image from a 2D image and the color table vectors.

You construct the 24-bit image by substituting the color table value into the appropriate plane of the 24-bit image. This is easily accomplished in IDL by subscripting the color table vector by the 2D image values. The code will look like this:

```
IDL> s = Size(image2d, /Dimensions)
IDL> image24 = BytArr(3, s[0], s[1])
IDL> TVLCT, r, g, b, /Get
IDL> image24[0,*,*] = r[image2d]
IDL> image24[1,*,*] = g[image2d]
IDL> image24[2,*,*] = b[image2d]
```

If the Display Depth is Greater than Eight

If the display depth is greater than eight, then you can obtain the 24-bit image directly. Be sure to set color decomposition on, however, to avoid the back-translation of the image value through the color table that you get on PC and Macintosh platforms.

```
IDL> Device, Decomposed=1
IDL> image24 = TVRD(True=1)
```

Writing the JPEG File

The final step, in either case, is to write the JPEG file, like this:

```
IDL> Write_JPEG, filename, image24, Quality=75, True=1
```

If you have an application that can open and read a JPEG file, try to read the one you just created. Many World Wide Web browsers support reading JPEG files. See if your browser has an *Open File* button with which you can try to read this JPEG file.

Reading a JPEG File

To read and display a JPEG file in IDL, use the *Read_JPEG* command, like this:

```
IDL> Read_JPEG, filename, thisImage
```

The variable *thisImage* is, of course, a 24-bit image and how it is displayed will depend on the depth of your display device. For example, if you are on an 8-bit display, you might display the graphic with these commands:

```
IDL> image2d = Color_Quan(thisImage, 1, r, g, b)
IDL> TVLCT, r, g, b
IDL> TV, image2d
```

Or, you can do this color quantization directly when you read the data from the file with keywords to the *Read_JPEG* command, like this:

```
IDL> Read_JPEG, filename, image2d, colortable, $
      Colors=!D.Table_Size, Dither=1, /Two_Pass_Quantize
```

The *colortable* parameter is an *N*-by-3 array containing the resulting color table vectors. The *Colors* keyword indicates how many colors the 24-bit image should be quantized into. It should be a value from 8 to 256. The *Dither* keyword selects the Floyd-Steinberg dithering method, which distributes the color quantization error to surrounding pixels and results in high-quality results. The *Two_Pass_Quantize* keyword makes the color quantization a two-step process, which also results in better color quantization and higher image quality.

To display the data, type these commands:

```
IDL> Erase
IDL> TVLCT, colortable
IDL> TV, image2d
```

If you are displaying the 24-bit image on a 24-bit display, then you can just read the file and display the graphic directly:

```
IDL> Read_JPEG, filename, thisImage
IDL> Device, Decomposed=1
IDL> TV, thisImage, True=1
```



Note that neither the *Device* command nor the *True* keyword is necessary with *TVImage*, since the program automatically determines whether it is displaying an 8-bit or 24-bit image and sets the correct color decomposition value and *True* keyword.

```
IDL> Read_JPEG, filename, image24
IDL> TVImage, image24
```

Note, too, that if I use the *TVImage* command, I don't even have to worry the depth of my display screen. *TVImage* takes all of this into account.

Creating Color TIFF Files

The process of creating color TIFF files is identical to the process of creating color JPEG files, with one important exception. (You can find the process described in “Creating Color JPEG Files” on page 154.) The exception is that when TIFF files created in IDL are displayed in other applications they are often displayed upside-down. There is an *Order* parameter on the *Write_TIFF* command (corresponding to the *Order* keyword used on a *TV* command or the *!Order* system variable) that can supposedly be used to change the order in which the image is displayed, but I have found that this value is rarely effective in practice. For this reason, I am in the habit of physically flipping the Y values of the image before I write the output to a file. This is easily done in IDL with the *Reverse* command.

For example, I almost always obtain or create a 24-bit image that is pixel-interleaved, meaning that it is a *3*-by-*m*-by-*n* array. I wish to flip the Y dimension of this image, which is the third image dimension. Suppose I wish to read the JPEG image I created above and save it as a TIFF file in IDL. I can type these commands:

```
IDL> Read_JPEG, filename, image24
IDL> basename = StrMid(filename, 0, StrLen(filename)-4)
IDL> Write_TIFF, basename + '.tif', Reverse(image24,3), 1
```

Opening the file in, for example, Photoshop results in the image being displayed upright.

Reading Dicom Image Files

I include a short introduction to reading DICOM Part 10 image files because the way it is done is slightly different (the functionality is written as an object) than the other file formats and because I think we will see more of this kind of functionality in the future. (DXF files are also read using an object.) Also, there are some limitations in DICOM files that you should know about. The first important limitation—as of IDL 5.3—is that you can only read DICOM files in IDL, you cannot write them.

First of all, even though the DICOM functionality is implemented in IDL as an object, you may not have to know this to read a simple DICOM image file. For example, if you are only interested in the images in a file, and not the other metadata, such as patient name, medical modalities, details of how the machine was calibrated, etc., then you can simply use the *Query_Dicom* and *Read_Dicom* commands to inquire about and read the images.

For example, suppose you want to read the *mr_knee.dcm* image file distributed with IDL. You can type these commands:

```
IDL> filename = Filepath('mr_knee.dcm', $
    SubDir=['examples', 'data'])
IDL> ok = Query_Dicom(filename, fileInfo)
IDL> IF ok THEN Help, fileInfo, /Structure

** Structure <1366108>, 7 tags, length=36, refs=1:
  CHANNELS      LONG           1
  DIMENSIONS    LONG          Array [2]
  HAS_PALETTE   INT            0
  NUM_IMAGES    LONG           1
  IMAGE_INDEX   LONG           0
  PIXEL_TYPE    INT            2
  TYPE          STRING         'DICOM'
```

You see that there is just a single image in this data file. Some DICOM files contain multiple images. For example, I've seen DICOM files that contain a small thumbnail image (usually 64-by-64) and the larger actual image. In that case, the *Dimensions* field only shows the dimensions of the first image, not the second. Although the *Num_Images* field will indicate that there are two images present. You can read the second image by setting the *Image_Index* keyword to 1 in the *Read_Dicom* command below.

To read and display this image, type these commands:

```
IDL> image = Read_Dicom(filename, Image_Index=0)
IDL> Window, XSize=fileInfo.Dimensions[0], $
    YSize=fileInfo.Dimensions[1]
IDL> LoadCT, 0
IDL> TV, image
```

Using the IDLffDicom Object

To access the actual metadata of the DICOM file, rather than just the image data, you must use the *IDLffDicom* object. You will learn more about objects (including how to

build your own objects) in a later chapter. But for now, just type the following command to create a DICOM object:

```
IDL> thisObject = Obj_New('IDLffDicom', filename)
```

To see the metadata of the file (or what is sometimes called the data dictionary), you can dump the file elements with the *DumpElements* method. The command looks like this:

```
IDL> thisObject->DumpElements
```

This particular file has 93 different data elements, and I don't want to show them all, but the first 25 elements in the file are shown in Figure 73.

0 : (0002,0000) : UL : META Group Length : 4 : 176
1 : (0002,0001) : OB : META File Meta Version : 2 : 0 1
2 : (0002,0002) : UI : META Media Stored SOP Class UID : 26 : 1.2.840.10008.5
3 : (0002,0003) : UI : META Media Stored SOP Instance UID : 44 : 1.2.840.113619.2
4 : (0002,0010) : UI : META Transfer Syntax UID : 18 : 1.2.840.10008.1.2
5 : (0002,0012) : UI : META Implementation Class UID : 18 : 1.2.840.113619.6.5
6 : (0002,0013) : SH : META Implementation Version Name : 6 : 1_2_5
7 : (0002,0016) : AE : META Source Application Entity Title : 6 : sdc21
8 : (0008,0000) : UL : ID Group Length : 4 : 390
9 : (0008,0001) : UL : ID Length to End (RET) : 4 : 132456
10 : (0008,0008) : CS : ID Image Type : 16 : ORIGINAL\PRIMARY
11 : (0008,0016) : UI : ID SOP Class UID : 26 : 1.2.840.10008.5.1.4.1.1.4
12 : (0008,0018) : UI : ID SOP Instance UID : 44 : 1.2.840.113619.2.1.2.139348932
13 : (0008,0020) : DA : ID Study Date : 8 : 19890203
14 : (0008,0021) : DA : ID Series Date : 8 : 19890203
15 : (0008,0023) : DA : ID Image Date : 8 : 19890203
16 : (0008,0030) : TM : ID Study Time : 6 : 092618
17 : (0008,0031) : TM : ID Series Time : 6 : 095819
18 : (0008,0033) : TM : ID Image Time : 6 : 095846
19 : (0008,0050) : SH : ID Accession Number : 0 :
20 : (0008,0060) : CS : ID Modality : 2 : MR
21 : (0008,0070) : LO : ID Manufacturer : 18 : GE MEDICAL SYSTEMS
22 : (0008,0080) : LO : ID Institution Name : 28 : THOMAS JEFF UNIVHOSPITAL MRI
23 : (0008,0090) : PN : ID Referring Physician's Name : 4 : HUME
24 : (0008,1010) : SH : ID Station Name : 8 : FOR.ICO
25 : (0008,1030) : LO : ID Study Description : 4 : KNEE

Figure 73: The first 25 data elements in the MR_Knee.dcm data file, obtained from dumping the elements to the command log window.

The first column of numbers is the *Reference Number*. The second column of numbers, which are enclosed by parentheses are the *Group* and *Element* references. These are expressed in hexadecimal notation. The third column of letters is the *Value Representation*. And what follows on each line is the *Description* of the data element.

The easiest way to obtain one of these data elements is to use its *Group* and *Element* number with the *GetValue* method of the object. The return value of this function will be a pointer array, with each pointer pointing to a data element having that particular *Group* and *Element* number. For example, to obtain the modality of this image, you can look at data element 20, which has a *Group* number of 0008 and an *Element* number of 0060, like this:

```
IDL> modality = thisObject->GetValue('0008'x, '0060'x)
IDL> Help, modality
```

```
MODALITY      POINTER      = Array [1]
```

Notice the way the *Group* and *Element* numbers are written as hexadecimal numbers in IDL. You access the element by de-referencing the pointer, like this:

```
IDL> FOR j=0,N_Elements(modality)-1 DO Print, *modality[j]
```

You can access other data elements in a similar fashion. For example, to obtain the image data element, which always has a *Group* number of 7FE0 and an *Element* number of 0010, you type this:

```
IDL> imagePtr = thisObject->GetValue('7FE0'x, '0010'x)
```

And to display the image you can type this:

```
IDL> Window, XSize=400, YSize=400  
IDL> TVImage, *imagePtr[0]
```

When you are finished with objects they should be destroyed.

```
IDL> Obj_Destroy, thisObject
```

But this doesn't destroy the pointers you might have created. These should also be freed up when you are finished with them. Otherwise, you can have memory leakage on the heap.

```
IDL> Ptr_Free, modality  
IDL> Ptr_Free, imagePtr
```

Chapter 6

♦ Discovering the Possibilities ♦♦♦



Reading and Writing HDF Data

Chapter Overview

HDF stands for *Hierarchical Data Format*. This is one of the three data formats that are called Scientific Data Formats in IDL. (The others are CDF and netCDF.) The HDF file format is a multi-object file format for sharing scientific data in a distributed environment. The format was developed at the National Center for Supercomputing Applications. The purpose of the HDF format is to provide a machine-independent, efficient data storage format for the various types of data and metadata (information that describes the data) commonly used by scientists. The HDF format has been chosen as the data file format for the NASA EOS (Earth Observing System) program because of the ease of retrieving, visualizing, analyzing and managing data that is stored in this format. Many data products, especially those coming from satellite systems, are now stored in the HDF format. IDL supports version 4.1r3 of the HDF library as of IDL 5.3.1. HDF-EOS support was also added in the IDL 5.2 version.

If you want to learn more about the HDF format, you can contact the National Center for Supercomputing Applications directly. Use your favorite World Wide Web browser to select the HDF home page at this URL:

<http://hdf.ncsa.uiuc.edu/>

The HDF Frequently Asked Questions (FAQ) file can be found at this World Wide Web address:

http://hdf.ncsa.uiuc.edu/HDF_FAQ.html

You will find a lot of good HDF information at this site, including HDF user guides and other HDF documentation.

In this chapter, you will learn:

- How the HDF file format is implemented in IDL
- How to read and write scientific data sets (SDS) in the HDF format
- How to store metadata with your scientific data sets
- How to store color table palettes with your scientific data sets

Why Use the HDF Format?

In today's world, scientists generate and process data from many different sources. They work on different computers. They use different kinds of software. They think of data differently. One scientist may be interested in data that is physically stored in separate files, but is related conceptually. Another may wish to consider only the data stored in a particular format in a particular file.

HDF addresses these problems by describing a format by which virtually any kind of data can be stored in a machine-independent way in one or many separate files. The HDF file format is supported on most of the computer operating systems scientists use today. HDF files allow different types of data to exist in the same file. For example, it is possible to have symbolic, numerical, and graphical data within the same HDF file. Simply put, the HDF file format allows scientists to communicate about science and not about the details of file sharing.

But more than that, HDF files are *self-describing*. For every piece of data in an HDF file, there can be metadata associated with it. Metadata is information that describes the data itself. It can be such information as how the instrument that collected the data was calibrated, what the scales for the axes should be, what dimensions should be used with the data, or where the actual data is stored (it can be in a physically separate file). This makes it possible to work with data that was collected by colleagues who may have long since retired or moved to other jobs.

Primary HDF Data Objects

A piece of data in an HDF file is said to be a *data object*. A data object contains two parts: a *data descriptor* and a *data element*. The data descriptor contains information about the type, location, and size of a data element. The data element contains the data itself. All data descriptors are 12 bytes long and they contain four fields. You see an illustration of a data descriptor in Figure 74, below.

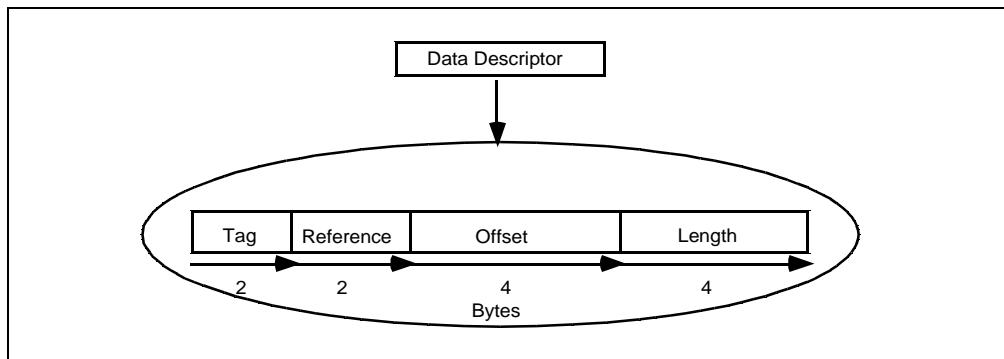


Figure 74: Each data object in HDF consists of a data descriptor, shown here, and a data element, which is the data itself.

The tag field identifies the type of data stored in the data element. There are over 200 tags defined by the NCSA for general use. You can get a complete list of tags from their World Wide Web site. You can see a short list of tags you might encounter in Table 11, below.

The reference number in the data descriptor distinguishes between different data elements with the same tag. For example, all scientific data sets will have the same tag, but the combination of tag number and reference number will uniquely determine a specific SDS in a file. The HDF routines will keep track of reference numbers for you as you write HDF data objects to a file.

Tag Number	Meaning
200	8-Bit Image Dimensions
201	8-Bit Palette
202	8-Bit Raster Image
701	SD Dimension Record
702	SD Data
703	SD Scales
704	SD Labels
705	SD Units
706	SD Formats
707	SD Max/Min Values
708	SD Coordinates
731	SD Calibration Data
732	SD Fill Value
1962	Vdata Description
1963	Vdata
1965	Vgroup

Table 11: Common HDF tag numbers and their associated meanings.

The offset field points to the location of the data element in the file in bytes. The length field identifies the size of the data element in bytes.

In general, you don't need to know much about the data descriptor except that it exists. You will find yourself accessing the tag and reference number fields in various ways. It will help if you know what these numbers refer to and how they are used.

There are five primary data objects allowed in an HDF file. They are *raster images* (both 8-bit and 24-bit), color *palettes*, *scientific data* (which are multi-dimensional arrays and, in fact, could be 8-bit and 24-bit images), *annotations*, and *Vdata* (which are tables of data). You see the five primary HDF data objects illustrated in Figure 75, below. A sixth data object, the *Vgroup*, does not contain data, but is used to group the other five primary data objects within an HDF file. Conceptually, a *Vgroup* is like an IDL structure variable.

An HDF file consists of an HDF file header and any number of these data objects, each with its data descriptor and data element. Data objects are individually accessible in the file, even if they are included in larger sets or groups of data. In fact, a single data object (e.g., a palette) may be included in several data sets or groupings.

HDF Application Programming Interface

At a low level, HDF is a physical format for storing scientific data. At a high level, HDF is a collection of utilities for manipulating and viewing data objects in HDF files. These utilities are in the form of IDL procedures and functions, which in the aggregate

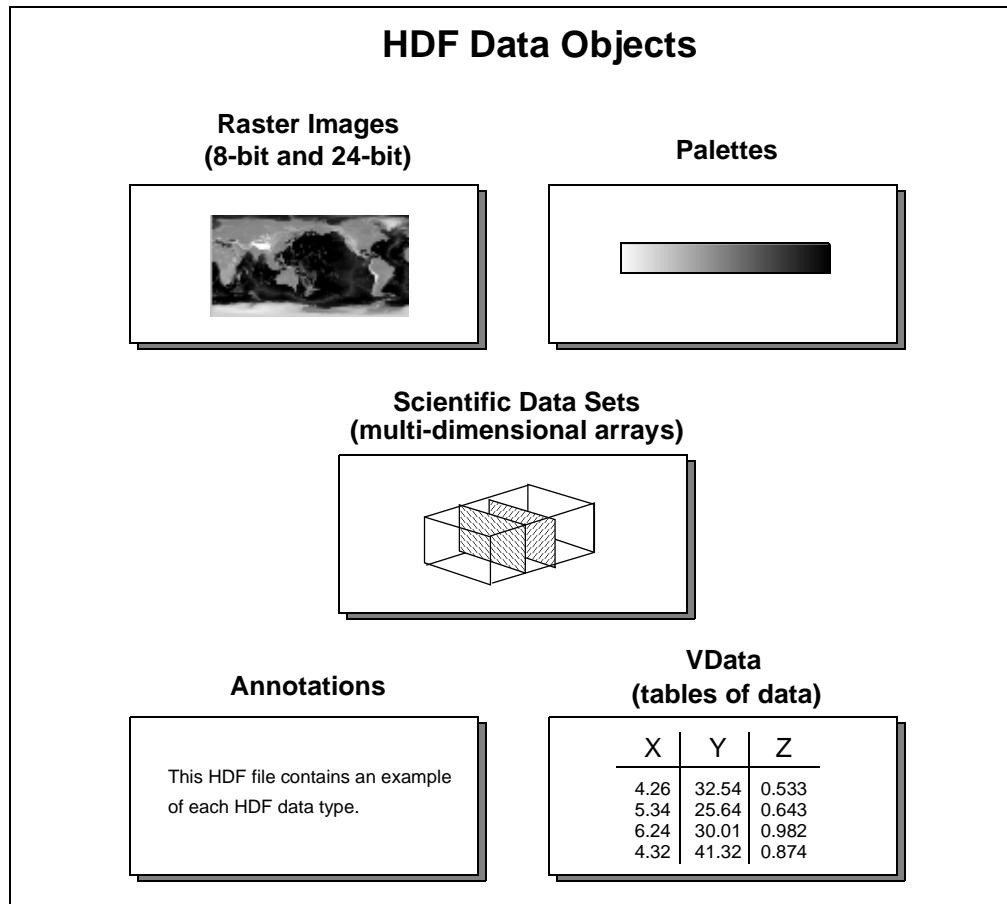


Figure 75: The five primary HDF data objects. A sixth object, the Vgroup, does not hold data, but serves to group any of these five primary objects within an HDF data file.

constitute what is called an *application programming interface* or API for HDF files. All the IDL routines in the HDF API start with the prefix *HDF_*. You can generally determine which data object the IDL routine applies to by the letters that follow this initial prefix. You see an example of this by viewing the IDL prefixes and the corresponding HDF objects the IDL routines apply to in Table 12, below.

Notice that there are two different APIs for working with scientific data. The older API uses the prefix *HDF_DFSD_*. This model was the original scientific data set (SDS) model developed by NCSA and it allowed access to a single HDF file at any one time. The new API uses the prefix *HDF_SD_* and supports a more powerful and general SDS model that allows access to multiple HDF files simultaneously. It also incorporates the netCDF data model developed by the Unidata Program Center. The *HDF_DFSD_* API was available in older versions of IDL mostly for backward compatibility with older HDF files. It has now been dropped altogether in IDL 5.3. The new *HDF_SD_* API should be used to create new HDF files.

Similarly, annotations should no longer be used in conjunction with SDS objects. Metadata that was once stored as annotations in the old SDS model is now more conveniently stored as attributes in the new SDS model. You will learn to use the new SDS API in this chapter.

HDF Data Object	IDL Prefix
24-bit raster image	HDF_DF24_
annotation data	HDF_DFAN_
palette data	HDF_DFP_
8-bit raster image	HDF_DFR8_
scientific data	HDF_SD_
Vdata	HDF_VD_
Vgroup	HDF_VG_

Table 12: *It is possible to determine which IDL routines apply to which HDF data objects by looking at the prefix of the IDL routines used in the HDF application programming interface.*

Working with HDF Files

The HDF programming model specifies that an HDF file be opened, manipulated, and then explicitly closed. To open and close files, you must be familiar with the HDF *file identifier*. HDF files are uniquely identified by either a file name or a file identification number, usually called the *fileID* or *fileHandle* in IDL documentation and code.

Opening HDF Files

For example, to open a new HDF file for writing, you might type:

```
fileID = HDF_Open(filename, /Create, /Write)
```

In this command the variable *filename* is a string with the name of the file. It is often obtained in a machine-independent way by using an IDL command like *Dialog_Pickfile*, like this:

```
filename = Dialog_Pickfile(/Write)
fileID = HDF_Open(filename, /Create, /Write)
```

The keyword *Create* opens a new file instead of one that already exists. The keyword *Write* puts the file in write access mode.

You may want to be sure that the file is an HDF file before you try to open it. All HDF files have a “magic cookie” or special number written as the first four bytes of the file. The HDF magic cookie is the hexadecimal number 0e031301. You can use the *HDF_IsHDF* command to read this magic cookie and tell you whether the file is an HDF file. The function returns a 1 if the file is an HDF file, and a 0 if not. You can use it like this:

```
filename = Pickfile(/Write)
IF HDF_IsHDF(filename) THEN $
    fileID = HDF_Open(filename, /Write, /Read)
```

Notice that the *Write* and *Read* keywords are both set in this open command. In this access mode, you can both read data from the file and write additional information to it as well. (An alternative keyword is *RdWr*, which has the same effect.)

Closing HDF Files

You will see how to read and write data to HDF files in just a moment. But to close an HDF file that was opened with the *HDF_Open* command, you just use the *HDF_Close* command, like this:

```
HDF_Close, fileID
```

Determining the Number of Tags in an HDF File

Once you have an HDF file open, you will want to know something about what is in the file. You will learn several ways to collect information from the HDF file in this chapter, but one of the ways you can learn about what is in the file is by determining how many tags are in the file. (Remember that each data object has a data descriptor with a tag field.)

You can imagine that if you opened an HDF file and knew absolutely nothing about it one way to proceed would be to find out how many tags or data descriptors were in the file, go to each one in turn, and use other API routines to inquire about each data object. Fortunately, this is not necessary usually. Most of the time you will have a very good idea about what is in an HDF file.

To determine how many tags are in an HDF file, you might use the *HDF_Number* command, like this:

```
fileID = HDF_Open(filename)
numberOfTags = HDF_Number(fileID)
```



Note that the number of tags in an HDF file is not very useful. In the first place, the number of tags will probably not be the same as the number of data objects you wrote to the file because there are a number of data descriptors written for each data object. These associated data descriptors pertain to attributes of data objects that may or not be filled in or written explicitly by the user. It is much more useful to know how many objects with a specific tag number are in the file.

Suppose you are interested in a particular type of data object, perhaps the number of SDS objects in the file. You can see from Table 11 on page 163 that if you wanted to know how many SDS objects were in the HDF file, you could look specifically for tag number 702. You do this with the *Tag* keyword, like this:

```
fileID = HDF_Open(filename)
numberOfSDSObjects = HDF_Number(fileID, Tag=702)
```

Working with Scientific Data Set HDF Files

The most popular API for HDF files is the SD or scientific data set interface. A scientific data set (SDS) is any multidimensional array. Since this kind of data is used extensively, it is a natural way of storing and manipulating data from within IDL. You can see a synopsis of the IDL SD API in Table 13, below.

Each SDS is associated with a dimension record and a data type. The SDS data model supports dimension *scales* as well as predefined and user-defined *attributes*. When an SDS object is created the number and size of the dimensions that define its shape are specified as well as the array's data type. The SDS object has an SDS *name*, which is a case-sensitive string. Names are assigned when the SDS is created and cannot be changed. The HDF interface will assign a name if you don't supply one when the SDS is created. SDS names do not have to be unique in an HDF file, but if they are not it will be difficult to sort out which SDS you want to interact with.

SDS dimensions specify the size and shape of a multidimensional array. The number of dimensions of the SDS array is known as the *rank* of the SDS. If you assign the same dimension name to two dimensions, the SD interface treats both dimensions as the same data object and any changes made to one will be reflected in the other. The size of a dimension will always be a positive integer.

Optional Dimension Scales

Your SDS can optionally have a dimension scale associated with it. A dimension scale is a sequence of numbers that divides the dimension into intervals. For example, a 2D array of temperature values might have two dimension scales associated with it. One might represent the longitudinal values of the 2D grid and the other might represent the latitudinal values.

Optional User-Defined Attributes

Your SDS can also have optional user-defined attributes. Attributes contain auxiliary information about a file or SDS. They are the so-called *metadata* associated with HDF files. It is data that describes the data. You can define any number of attributes.

Attributes can be attached to three types of objects: files, data sets, and dimensions. These are referred to, respectively, as *file attributes*, *data set attributes*, and *dimension attributes*.

File Attributes These attributes describe an entire file. They generally contain information about the data in the file as a whole. They are sometimes referred to as *global attributes*.

Data Set Attributes These attributes describe features of the individual SDS. Because their scope is limited to a specific SDS, they are often called *local attributes*.

Dimension Attributes These attributes describe features of the individual SDS dimensions. For example, the unit of a dimension is one of its attributes.

Each object has its own attribute count, which identifies the number of attributes associated with the object. The attribute count begins at 0 and is increased by one for each additional attribute associated with an object. Thus, each attribute has its own “index”, which is used to retrieve the information contained in the attribute. Attribute names are treated like dimension names, so you will want to provide meaningful names for your attributes.

The IDL SD interface uses the same commands to access file and SDS attributes, so you have to be careful to use the proper identifier. File identifiers access file attributes and SDS identifiers access data set attributes. Dimension attributes are set with a separate command.

Optional Predefined Attributes

SDS data sets and dimensions can be assigned optional predefined attributes. For example, data sets and dimensions may have these predefined attributes:

Format A string that contains the format for the object data. The format will be used for printing or display of the data.

Label A label can be thought of as an independent variable name for the object. Labels are often used as search keys.

Unit Units specify what the numbers associated with the data mean.

Category	IDL Routine	Type	Description
Access	HDF_SD_End	P	Closes the HDF file and cleans up memory.
	HDF_SD_EndAccess	P	Terminates the connection to the SDS.
	HDF_SD_Select	F	Establishes connection to an SDS via its SDS identifier.
	HDF_SD_Start	F	Opens HDF file and establishes the SD interface.
Read/Write	HDF_SD_AddData	P	Writes a slab of data to the SDS.
	HDF_SD_GetData	P	Reads a slab of data from the SDS.
	HDF_SD_Create	F	Creates a new SDS and returns SDS ID identifier.
	HDF_SD_SetTextFile	P	Moves SDS data to an external file.
General Inquiry	HDF_SD_FileInfo	P	Reads information about the HDF file contents.
	HDF_SD_GetInfo	P	Reads information about the SDS.
	HDF_SD_IDtoRef	F	Given an SDS ID, returns the SDS reference number.
	HDF_SD_IsCoordVar	F	Will identify netCDF “coordinate” variables.
	HDF_SD_NameToIndex	F	Given the SDS name, returns the SDS index number.
	HDF_SD_RefToIndex	F	Given the SDS reference number, returns SDS index.
Dimensions	HDF_SD_DimGet	P	Reads the attributes of an SDS dimension object.
	HDF_SD_DimGetID	F	Returns the dimension ID.
	HDF_SD_DimSet	P	Writes the attributes of an SDS dimension object.
User Attributes	HDF_SD_AttrInfo	P	Reads information from file and SDS attributes.
	HDF_SD_AttrSet	P	Writes user-defined file and SDS attributes.
	HDF_SD_AttrFind	F	Given an attribute’s name, returns its index.
Predefined Attributes	HDF_SD_SetInfo	P	Writes predefined SDS attributes.

Table 13: *The IDL application programming interface (API) for HDF scientific data sets. The Type column identifies which routines are IDL procedures as opposed to IDL functions.*

In addition to these attributes, data sets may have these other predefined attributes:

Calibration	This attribute stores the scale and offset values used to calibrate the data in the SDS.
Coordinate System	This is the coordinate system associated with a particular data set (e.g, “Device” or “Normal”)
Fill Value	This is a value used to fill areas between non-contiguous writes to SDS arrays.
Range	A two-element vector that describes the minimum and maximum value of the data set.

Opening HDF Files Containing Scientific Data Sets

If the HDF file you are going to be working with contains scientific data sets (or you are only interested in the SDS data objects and information associated with them in the file), you can open the file and initiate the SDS API at the same time with the *HDF_SD_Start* command. For example, to open a new file for storing SDS objects, you can type:

```
sdFileID = HDF_SD_Start(filename, /Create)
```

To open an SDS file for reading and writing, you can type:

```
sdFileID = HDF_SD_Start(filename, /RdWr)
```

Closing HDF Files Containing Scientific Data Sets

Closing HDF files that were opened with the *HDF_SD_Start* command is relatively straightforward, too. You use the *HDF_SD_End* command. The only complication is that if you have created or selected any SDS objects in the course of data manipulation, you must terminate this access before you close the file. (You will learn how to create and select SDS objects with the *HDF_SD_Create* and *HDF_SD_Select* commands below.) If you don't terminate access to each SDS there is a chance the data in the file will be corrupted when the file is closed. Access to the SDS object is terminated with the *HDF_SD_EndAccess* command.

The correct sequence of open, manipulation, and close commands for SDS HDF files will look something like this:

```
sdFileID = HDF_SD_Start(filename, /RdWr)
newSDS_ID1 = HDF_SD_Create(sdFileID, 'New Data', $[100L], /Float)
oldSDS_ID1 = HDF_SD_Select(sdFileID, 3)
. . .; Other SDS manipulation commands here.
HDF_SD_EndAccess, newSDS_ID1
HDF_SD_EndAccess, oldSDS_ID1
HDF_SD_End, sdFileID
```

Creating or Selecting Scientific Data Sets

Once you have opened an HDF file and initiated the SD interface with the *HDF_SD_Start* command, you must either create or select an SDS to work with. You will use the *HDF_SD_Create* command to create a new SDS, or the *HDF_SD_Select* command to select an SDS that already exists in the file. Both of these commands will return an SDS identifier or handle, which is what you will use to identify and work with this SDS.

Creating a New SDS

To create a new SDS in the HDF file, you will need to give the SDS a name and indicate the size and number of dimensions in the SDS array. You will also indicate with a keyword the SDS data type. For example, suppose you have a 2D image that is 300 by 400 in size and contains floating-point data and you wish to store this in an SDS. You create the SDS identifier for this SDS with the *HDF_SD_Create* command like this:

```
sdFileID = HDF_SD_Start(filename, /Create)
sdsID = HDF_SD_Create(sdFileID, 'image', [300, 400], /Float)
```

To write the data into the SDS, you use the *HDF_SD_AddData* command, like this:

```
array = FltArr(300, 400)
```

```
HDF_SD_AddData, sdsID, array
```

Suppose you had three of these 300 by 400 floating-point arrays, and you wanted to store them in the same SDS. You might create the SDS like this:

```
sdsID = HDF_SD_Create(sdFileID, 'image', $  
[300, 400, 3], /Float)
```

To write an array into the second position in this SDS, you can use the *Start* keyword with the *HDF_SD_AddData* command, like this:

```
HDF_SD_AddData, sdsID, array, Start=[0,0,1]
```

To read the array out into another IDL variable, *thisArray*, use the *HDF_SD_GetData* command. You will have to use both the *Start* and *Count* keywords to get the correct information out of the file. The *Start* keyword indicates where to start reading in the SDS. The *Count* keyword indicates how much information to read. Your code will look like this:

```
HDF_SD_GetData, sdsID, thisArray, Start=[0,0,1], $  
Count=[300, 400, 1]
```

It is even possible in an HDF data set to have an “unlimited” dimension. For example, suppose you didn’t know off-hand how many of these 300 by 400 floating point arrays you wanted to add to this SDS. If the last dimension of the dimension record of an SDS is a negative number (or 0) then this dimension is considered “unlimited”. For example, you could create the SDS like this:

```
sdsID = HDF_SD_Create(sdFileID, 'image', $  
[300, 400, -1], /Float)
```

Now, you can write as many of these arrays as you like into this SDS. For example, you can write something like this:

```
bigData = FltArr(300, 400, 10)  
HDF_SD_AddData, sdsID, bigData
```

Later on, you could add more data to this SDS, like this:

```
HDF_SD_AddData, sdsID, array, Start=[0, 0, 10]
```

Selecting an Existing SDS

To select an existing SDS in an HDF file, you will use the *HDF_SD_Select* command. Each SDS is assigned an index number, starting with index 0, in the order in which they are created in the HDF file. The *HDF_SD_Select* command returns the ID of the SDS. For example, if you wanted to read and write to the fourth SDS in the file *exp12h.hdf*, you might type this:

```
sdFileID = HDF_SD_Start('exp12h.hdf', /RdWr)  
sdsID = HDF_SD_Select(sdFileID, 3)
```

Adding Attributes to Scientific Data Sets and HDF Files

Attributes are the metadata that are associated with the actual data of an HDF file. Attributes can be assigned to the HDF file itself, to SDS objects in the file, or to the dimensions of an SDS. You can make up attributes yourself (these are called user-defined attributes) or you can use a number of predefined attributes. File and SDS attributes are assigned with the *HDF_SD_AttrSet* and *HDF_SD_SetInfo* commands. Dimension attributes are assigned with the *HDF_SD_DimSet* command.

Suppose you are writing a new HDF file. You might like to include in the file the name of the experiment that produced the data, the date of the experiment, who performed the experiment, and maybe how to get in touch with that person. All of this

user-defined information can be stored with the HDF file itself as attributes by using the *HDF_SD_AttrSet* command. Your code might look something like this:

```

hdfFileID = HDF_SD_START('newexp.hdf')

date = '28 January 1997'
experiment = '1-28-97GH45'
scientist = 'Mary McGuire'
email = 'mary@someuni.edu'

HDF_SD_SetAttr, hdfFileID, 'EXPERIMENT DATE', date
HDF_SD_SetAttr, hdfFileID, 'EXPERIMENT NAME', experiment
HDF_SD_SetAttr, hdfFileID, 'EXPERIMENTER', scientist
HDF_SD_SetAttr, hdfFileID, 'EMAIL ADDRESS', email

```



Notice that the names of the attributes are capitalized here. This is not a requirement, but the names will be case sensitive. It will be easier to work with attributes later if you are consistent in your use of case. Also note that these strings should not be longer than 256 characters, or it may not be possible to retrieve the entire attribute from the file.

Suppose this experiment resulted in the acquisition of two temperature variables, each a floating-point array of 100 by 200 in size. Suppose one variable is named *temp12* and the other is named *temp24*.

You might create SDS objects for these two data sets and load them like this:

```

sdsID12 = HDF_SD_Create(hdfFileID, '12 Hour Temperature', $
    [100, 200], /Float)
HDF_SD_AddData, sdsID12, temp12
sdsID24 = HDF_SD_Create(hdfFileID, '24 Hour Temperature', $
    [100, 200], /Float)
HDF_SD_AddData, sdsID24, temp24

```

You might want to attach information about these data sets directly to data itself. For example, you might want to include the predefined attributes *Range* and *Unit* to these two data sets. You use the *HDF_SD_SetInfo* command to set predefined attributes. Your code might look like this:

```

HDF_SD_SetInfo, sdsID12, RANGE=[-35, 50], Unit='DEGREES'
HDF_SD_SetInfo, sdsID24, RANGE=[-25, 56], Unit='DEGREES'

```

Perhaps the 12 hour data set had an anomaly associated with it. Maybe the instrument collecting the data malfunctioned. You might want to note this with a user-defined attribute for that data set. You might type this:

```

problem = 'Instrument Malfunction'
HDF_SD_SetAttr, sdsID12, 'PROBLEM', problem

```

Perhaps you even calculated the coefficients of an equation that you used to correct the data. Suppose those coefficients were stored in a 4-element variable named *correct_coeff*. You could store these with the 12 hour data set like this:

```

HDF_SD_SetAttr, sdsID12, 'CORRECTION COEFFICIENTS', $
    correct_coeff

```

Finally, suppose that these two data sets had vectors associated with them that described the range of longitude (a 100-element vector) and latitude (a 200-element vector associated with the temperature grid. Suppose these vectors were named *lon* and *lat*, respectively. You might want to store these as the dimensions of the two temperature data sets.

To do this, you will need access to the dimension objects associated with the data sets. You can obtain this access with the *HDF_SD_DimGetID* command. There are two

dimension objects associated with each SDS. The dimension objects are identified by dimension index numbers that begin with 0 and increase by one for each dimension of the SDS (as defined above with the *HDF_SD_Create* command).

You can get the dimension object IDs for the 12 hour data set by typing this:

```
lon12dimID = HDF_SD_DimGetID(sdsID12, 0)
lat12dimID = HDF_SD_DimGetID(sdsID12, 1)
```

Now you can store the *lon* and *lat* vectors in the dimension objects with the *HDF_SD_DimSet* command. At the same time, you will set some of the predefined attributes for the dimension objects. Type this:

```
HDF_SD_DimSet, lon12dimID, $
LABEL='Longitude', $
NAME='Longitude Axis', $
SCALE=lon, $
UNIT='Degrees'

HDF_SD_DimSet, lat12dimID, $
LABEL='Latitude', $
NAME='Latitude Axis', $
SCALE=lat, $
UNIT='Degrees'
```

You can do something similar for the 24 hour data set, but this time you will take advantage of the fact that dimensions with the same name, even if they are assigned to a different SDS, will point to the same dimension object. Type this:

```
lon24dimID = HDF_SD_DimGetID(sdsID24, 0)
lat24dimID = HDF_SD_DimGetID(sdsID24, 1)
HDF_SD_DimSet, lon24dimID, NAME='Longitude Axis'
HDF_SD_DimSet, lat24dimID, NAME='Latitude Axis'
```

Gathering Information about Scientific Data Sets

The most common type of HDF file is one containing any number of scientific data sets with their associated attributes. The best way to learn about this kind of HDF file is to use the *HDF_SD_FileInfo* command. The command can be used along with the *HDF_SD_Start* command to find out how many scientific data sets and file attributes are in the file. In the *HDF_SD_FileInfo* command below, the variables *datasets* and *attributes* are output IDL variables that contain the number of datasets and the number of file attributes, respectively, that have been defined for the file. The commands might look like this:

```
sdFileID = HDF_SD_Start(filename, /RdWr)
HDF_SD_FileInfo, sdFileID, datasets, attributes
```

These data sets and attributes have names, which you can retrieve and use to manipulate the data and attributes associated with it. The data sets and attributes are accessed by means of their zero-based index position in the file, but in different ways. For data sets you use the *HDF_SD_Select* command to get a data set ID and then use this ID with the *HDF_SD_GetInfo* command to find the name of the data set. For example, to get the name of the first data set in an HDF file (index 0), you can type this:

```
sdFileID = HDF_SD_Start(filename, /Read)
thisSDS = HDF_SD_Select(sdFileID, 0)
HDF_SD_GetInfo, thisSDS, Name=thisSDSName
Print, thisSDSName
```

If you want the name of the first attribute in the file, you do it directly without first getting a data set ID. You will use the *HDF_SD_AttrInfo* command, like this:

```
sdFileID = HDF_SD_Start(filename, /Read)
HDF_SD_AttrInfo, sdFileID, 0, Name=thisAttrName
Print, thisAttrName
```

Suppose you have an HDF file, but you don't know anything about the number of data sets or attributes that are in the file. You could open the file and print the names of all the data sets in the file, by writing code like this:

```
sdFileID = HDF_SD_Start(filename, /Read)
HDF_SD_FileInfo, sdFileID, datasets, attributes
FOR j=0, datasets-1 DO BEGIN
    thisSDS = HDF_SD_Select(sdFileID, j)
    HDF_SD_GetInfo, thisSDS, Name=thisSDSName
    Print, 'Dataset No. ', StrTrim(j, 2), ': ', thisSDSName
ENDFOR
```

Having established all the data set names, you can now print out all the attribute names, with code like this:

```
FOR j=0, attributes-1 DO BEGIN
    HDF_SD_AttrInfo, sdFileID, j, Name=thisAttr
    PRINT, 'Attribute No. ', + StrTrim(j, 2), ': ', thisAttr
ENDFOR
```

Gathering SDS Attribute Information

There is a comparable command to *HDF_SD_FileInfo* that can be used to find the number of attributes attached to an SDS data set. It is the *HDF_SD_GetInfo* command. The code to get all of the SDS data sets in the file and print the names of all the attributes attached to them will look like this:

```
sdFileID = HDF_SD_Start(filename, /Read)
HDF_SD_FileInfo, sdFileID, datasets
FOR j=0, datasets-1 DO BEGIN
    thisSdsID = HDF_SD_Select(sdFileID, j)
    HDF_SD_GETINFO, thisSdsID, NATTS=numAttributes
    FOR j=0,numAttributes-1 DO BEGIN
        HDF_SD_ATTRINFO, thisSdsID, j, NAME=thisAttr
        PRINT, 'SDS Attribute No. ', + STRTRIM(j, 2), ' : ', $
            thisAttr
    ENDFOR
ENDFOR
```

Adding Color Palettes to HDF Files

Color palettes are a device for specifying colors for an HDF data set. Another term for a color palette might be a *color lookup table*. Color palettes in many applications, IDL included, may be any size, but HDF files only support color palettes that consist of red, green, and blue vectors of 256 elements in size. These color vectors, which together specify the (r,g,b) color triple associated with a pixel value, must reside in a palette array that is dimensioned 3 by 256.

It is easy to include a color palette with an HDF file in IDL. The color palette is written to the file with the *HDF_DFP_AddPal* command and read from the file with the *HDF_DFP_GetPal* command.

It is a good idea to load color tables and obtain the red, green, and blue vectors that compose those tables from within the Z-graphics buffer. The advantage of doing this is that you are assured of having 256 colors, no matter how many colors you are using in your IDL session. The code to produce the 3 by 256 array that is going to be stored as a color palette in the HDF file looks something like this (color table 5 is loaded in this example):

```
thisDevice = !D.NAME  
SET_PLOT, 'Z'  
LOADCT, 5, /SILENT  
TVLCT, r, g, b, /GET  
palette = BYTARR(3,256)  
palette(0,*) = r  
palette(1,*) = g  
palette(2,*) = b  
SET_PLOT, thisDevice
```

When you are ready to store the palette in the HDF file, type this:

```
HDF_DFP_ADDPAL, filename, palette
```

When you want to retrieve the color palette from the file, use this command:

```
HDF_DFP_GETPAL, filename, thisPalette
```

To use the color palette in IDL, it must be resized to the number of colors in your IDL session. The 3 by 256 array must also be transposed into a 256 by 3 array in order to accommodate the *TVLCT* command. Your code might look something like this:

```
TVLCT, Transpose(Congrid(thisPalette, 3, !D.Table_Size-1))
```

Examples of Reading and Writing HDF Files

The programs *HDFRead* and *HDFWrite* are among the programs you downloaded to use with this book. These are complete examples of the principles outlined in this chapter. You may also see the source code for these two programs in the IDL Source Code appendix on page 399.

Chapter 7

◆ Discovering the Possibilities ◆◆◆



Creating Hardcopy Graphics Output

Chapter Overview

In using IDL, there is perhaps no topic more difficult or as misunderstood as the question of how to reproduce what you see on your display screen in hardcopy form. And yet, this is the one requirement that almost all of us who pursue scientific endeavors must satisfy, for there are few other completely satisfactory ways to share our results among colleagues.

This chapter will concentrate on PostScript output, since PostScript is almost universally accepted as the output medium of choice and most programmers working with IDL will have access to a PostScript printer. Nearly all of what is said about PostScript also applies to other output devices such as HP plotters or PCL printers.

Specifically, you will learn in this chapter:

- How to select a hardcopy output device
- How to configure a hardcopy output device
- How to send graphical output directly to a printer
- How to send graphical output to a file
- How to create graphical output for the hardcopy output device
- How PostScript output differs from the output on your display
- How to position plots and images on the PostScript page
- How to produce graphical output that can be included in other documents
- How to write graphical programs that are easily converted to hardcopy output
- How to use color in PostScript output

Selecting the Graphics Hardcopy Output Device

You set the graphics hardcopy output device in IDL just like you set any other graphics display device, with the *Set_Plot* command:

```
Set_Plot, 'option'
```

where *option* is any one of the options listed below. Notice that *option* is always a string, so it is usually enclosed in quotes. Unlike many other strings in IDL, *option* is also case insensitive.

CGM	The output is written to a file in CGM, or Computer Graphics Metafile, format. CGM is a device independent file format used to exchange graphic information. CGM files can be encoded in one of three methods: (1) text, (2) binary, and (3) NCAR binary.
HP	The output is written to a file in Hewlett-Packard Graphics Language (HP-GL) format, suitable for output on a variety of HP-GL pen plotters.
PCL	The output is written to a file in Hewlett-Packard Printer Control Language (PCL) format, suitable for output on a variety of HP laser and inkjet printers.
PRINTER	The output is sent directly to the default printer in whatever form is appropriate for the printer.
PS	The output is written to a file in PostScript format.
Z	The output is written into the Z-graphics buffer.

After printing, you should set the output device back to your type of graphics display device with the *Set_Plot* command. Here are the usual display devices.

WIN	A personal computer running the Microsoft Windows or NT operating system.
MAC	A computer running the MacOS operating system.
X	A computer running an X Window windowing system.

Only one device can be your *current* graphics device. You can determine which device is current by examining the *!D.Name* system variable, like this:

```
IDL> Print, !D.Name
```



Note that the name of the device is case insensitive when you set it, but it is *not* case insensitive when you *use* the name in your code. The graphics device name that is stored in *!D.Name* is always stored in uppercase characters. This is vitally important in string comparison statements like this one:

```
IDL> IF !D.Name EQ 'PS' THEN Print, 'Using PostScript...'
```

Configuring the Graphics Hardcopy Output Device

Once you select the hardcopy output device, all device-specific configuration parameters are controlled by means of keywords to the *Device* command. The keywords that are available to the *Device* command depend upon which device is the current device. The *Printer* device (which is always associated with your default printer) may also be configured with the *Dialog_PrinterSetup* command, which is described in more detail in “Configuring and Using the Printer Device” on page 200.

Determining the Current Device Configuration

To see what configuration parameters are available (and currently selected) on the current hardcopy output device, use the *Help* command, like this:

```
IDL> Help, /Device
```

You will be shown a list of the available configuration parameters for your current graphics device and their current values. This is the information you will need to

configure the device. Depending upon which device you have set as your current device, you might see information about how many colors are available on that device, what graphics function IDL is using, what hardware font is currently selected, and so on.

Notice that this information display is different for each hardcopy output option. For example, type these commands to see how the PostScript output device is configured by default:

```
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PS'
IDL> Help, /Device
IDL> Set_Plot, thisDevice
```

Here is the result of the *Help* command, issued on a Windows NT machine:

```
Available graphics_devices: CGM HP NULL PCL PRINTER PS WIN Z
Current graphics device: PS
File: <none>
Mode: Portrait, Non-Encapsulated, EPSI Preview Disabled,
      Color Disabled
Offset (X,Y): (1.905,12.7) cm., (0.75,5) in.
Size (X,Y): (17.78,12.7) cm., (7,5) in.
Scale Factor: 1
Font Size: 12
Font Encoding: AdobeStandard
Font: Helvetica
# bits per image pixel: 4
(!3) Helvetica          (!4) Helvetica-Bold
(!5) Helvetica-Narrow   (!6) Helvetica-Narrow-Bold
(!7) Times-Roman        (!8) Times-BoldItalic
(!9) Symbol              (!10) ZapfDingbats
(!11) Courier            (!12) Courier-Oblique
(!13) Palatino-Roman    (!14) Palatino-Italic
(!15) Palatino-Bold      (!16) Palatino-BoldItalic
(!17) AvantGarde-Book   (!18) NewCenturySchlbk-Roman
(!19) NewCentury-Bold    (!20) Undefined-User-Font
```

Common Device Command Keywords

Most output devices allow the following keywords to be used with the *Device* command. (The *Z* device is an exception.) These are keywords you will want to know. You should consult the IDL on-line documentation for other keywords that may be used for a particular output device. For example, the *PS* device accepts nearly 50 different keywords.

Close_Document	This keyword closes the graphics document after flushing the output buffer. It is required to eject the document from the printer. (It is used with the <i>Printer</i> device.)
Close_File	This keyword closes the graphics output file after flushing the output buffer. (It is used with the <i>PS</i> device.)
Filename	The graphics output devices that write their output to a file have a default file name. This is the file in which the graphic output will be placed if no file name is specified. Normally the file is named <i>idl.option</i> , where <i>option</i> is the type of hard-copy output device you have selected. However, the name of the file can be changed by specifying it with this keyword. For example, like this:

```
IDL> Device, Filename='surface.eps'
```

Inches

If this keyword is set, the *XSize*, *YSize*, *XOffset*, and *YOffset* keywords and settings are assumed to be specified in inches rather than in the default unit of centimeters.

```
IDL> Device, XSize=4.0, /Inches
```



To set the size and offset settings back to centimeters, use:

```
IDL> Device, Inches=0
```

Landscape

If set, this keyword specifies that the output should be in landscape orientation on the page.

Portrait

If set, this keyword specifies that the output should be in portrait orientation on the page, the default.

XOffset

If set, this keyword specifies the X position, on the page, of the lower-left corner of the output display window (in portrait mode). For details on landscape mode, see “Calculating PostScript Offsets in Landscape Mode” on page 198.

XSize

If set, this keyword specifies the width of the output display window on the page.

YOffset

If set, this keyword specifies the Y position, on the page, of the lower-left corner of the output display window (in portrait mode). For details on landscape mode, see “Calculating PostScript Offsets in Landscape Mode” on page 198.

YSize

If set, this keyword specifies the height of the output display window on the page.

```
IDL> Device, XSize=4.0, YSize=7.0, /Inches
```



Note that once you use a keyword to configure a parameter on the graphics output device, that configuration parameter stays in effect until you explicitly change it or exit IDL.

The *XSize*, *YSize*, *XOffset*, and *YOffset* keywords are generally used to position a “graphics window” on the output page. IDL commands go into the graphics window in a way that is entirely analogous to the way graphics output goes into a graphics window located on the display. See “Similarities Between the Display and PostScript Devices” on page 184 for details.

Creating the PostScript File

The graphics device that is current and to which all graphics commands will be directed is always stored in the system variable *!D.Name*. So, in practice—especially in IDL programs—the code to select a hardcopy output device usually looks like the example here, in which data is created and sent to a PostScript file named *output.ps*. Notice how the device is selected, configured, and then closed after the graphics commands are written into the file. Closing the file is essential. If you forget to do this, you will not be able to view the file with a PostScript viewer, nor will you be able to print the file on your PostScript printer.



If you fail to close the file the printer will actually *process* the page, but it can’t *eject* the page from the printer because the file lacks the PostScript *showpage* command, which is required to make this happen. This can be disconcerting if you have a slow printer like I do and you are used to standing around drinking a cup of coffee while it chugs away. The first time this happened to me I went through two pots of coffee before I discovered what I was doing wrong.

```
IDL> data = LoadData(1)
IDL> thisDevice = !D.Name
```

```

IDL> Set_Plot, 'PS'
IDL> Device, Filename='output.ps', XSize=4, YSize=4, $
      /Inches, XOffset=2.25, YOffset=3.5

IDL> Plot, data
IDL> Device, /Close_File
IDL> Set_Plot, thisDevice

```

Sending Graphics to the Hardcopy Device

The general idea in producing hardcopy output is to select the hardcopy output device with the *Set_Plot* command, configure the device the way you want it with the *Device* command (or sometimes with *Dialog_PrinterSetup* in the case of the *Printer* device), and then execute the same IDL commands that would have sent the output to the display device. Instead, of going to the display device, however, the commands will be placed in a file or sent directly to the printer. When you are finished issuing IDL graphics commands, you close the output file or print job and route the file to the printer or plotter appropriate for the type of file you produced. In the case of the *Printer* device, the routing is done automatically for you. (See “Printing PostScript Files” on page 180 for more information about sending PostScript files produced by IDL to the printer.)

For example, to create a simple plot in a PostScript file, you might type these commands:

```

IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PS'
IDL> Device, Filename='plot.ps', XSize=4, YSize=4, $
      /Inches, XOffset=2.25, YOffset=3.5

IDL> Plot, LoadData(1)
IDL> Device, /Close_File

```

If you produced this PostScript file on a UNIX computer, you might route the file to the printer by issuing a simple *lpr* command (or whatever the equivalent local command is on your computer). For example, from within IDL you might type this command:

```
IDL> Spawn, 'lpr plot.ps'
```

In broad terms, the output you send to your display device and the output you send to the PostScript file is not very different. (The devil is in the details, they say.) That is, the *Plot*, *Surface*, *Contour*, and other IDL graphic commands work approximately the same whether they are being sent to the display or to a PostScript file.

One way they are similar is in how they go into the file. For example, if you issued a *Plot* or *Surface* command and your display was the current graphics device, you know that the contents of the current window would be erased and a new graphic display would be created in its place. Something similar happens when the PostScript device is the current graphics device. Each command that would erase the window on your display starts a new page of PostScript output.

Similarly, each graphics command, such as *OPlot* or *XYOutS*, that would be executed in the current display window modifies the current page of PostScript output. So, for example, to display a plot with a title written with *XYOutS* and a surface plot, you might type something like this:

```

IDL> Plot, LoadData(1), Position=[0.1, 0.1, 0.9, 0.8]
IDL> XYOutS, 0.5, 0.9, 'Simple Plot', Align = 0.5, /Normal
IDL> Surface, Dist(41)

```

If these commands were issued to the PostScript device instead of to the display device, you would have a file that contained two pages of output. To send these commands to a PostScript file, you might type commands like this:

```
IDL> thisDevice = !D.Name  
IDL> Set_Plot, 'PS'  
IDL> Device, XSize=3, YSize=3, /Inches  
IDL> Plot, LoadData(1), Position=[0.1, 0.1, 0.9, 0.8]  
IDL> XYOutS, 0.5, 0.9, 'Simple Plot', Align = 0.5, /Normal  
IDL> Surface, Dist(41)  
IDL> Device, /Close_File  
IDL> Set_Plot, thisDevice
```

Notice that you can use keywords like *Position* and *Normal* to place graphical display elements in either the display window or a PostScript window. This is a good way to create output that looks the same whether it is on the display or in a PostScript file. You will read more about this in a moment.



A trick you can use to force the PostScript file to advance a page is to use the *Erase* command. This is handy, for example, if you want to place several images in a PostScript file. Normally, the display window is not erased prior to a *TV* or *TVScl* command, so multiple *TV* commands will simply stack each image on top of the previous one. The *Erase* command allows you to put the images into the same file on different pages, like this:

```
IDL> thisDevice = !D.Name  
IDL> Set_Plot, 'PS'  
IDL> TV, LoadData(5)  
IDL> Erase  
IDL> TV, LoadData(7)  
IDL> Device, /Close_File  
IDL> Set_Plot, thisDevice
```

Printing PostScript Files

The easiest way to print PostScript files from a UNIX machine is to spawn the *lpr* command. (Or whatever the equivalent local command is on your machine.) But you might make the mistake of trying to print your PostScript file from within IDL with a series of commands that look like this:

```
Set_Plot, 'PS'  
Device, Filename='new_plot.ps'  
Plot, my_data, Title='Awful Nice Plot'  
Spawn, 'lpr new_plot.ps'
```

Unfortunately, this will not work, and the reason is that you forgot to close the PostScript file before you tried to print it with the UNIX *lpr* command. The correct sequence of commands is this:

```
Set_Plot,'PS'  
Device, Filename='new_plot.ps'  
Plot, my_data, Title='Awful Nice Plot'  
Device, /Close_File  
Spawn, 'lpr new_plot.ps'
```

The reason the first series of commands above do not work is that the PostScript file requires a PostScript *showpage* command to eject the file from the printer. The *showpage* command is not inserted into the file, however, until the file is closed by using the *Close_File* keyword or by exiting IDL.

On personal computers, it is a bit harder to print PostScript files created with the *PS* device from within IDL. In fact, most people don't bother, since there are numerous utilities for printing PostScript files available for these machines.

Printing PostScript Files on Computers Running MacOS

On a Macintosh or a computer running the MacOS operating system, the best way to send a PostScript file to the printer is to download the small, free program *Drop•PS* from Bare Bones Software, Inc. It is available for downloading from the usual Macintosh anonymous ftp software sites. Macintosh printers always come with printer utilities that also provide a way to send a PostScript file directly to the printer.

Printing PostScript Files on a Windows Computer

On a computer running a Windows operating system, the best way to send a PostScript file to the printer is to download the free software programs *GhostView* or *GhostScript*. These programs are PostScript viewers and are essential for viewing PostScript output before it is printed. *GhostView* has a wonderful graphical user interface that also allows you to send PostScript files to the printer. You can learn more about these programs by using a World Wide Web browser to look at their home page. The URL is:

<http://www.cs.wisc.edu/~ghost/>

GhostScript is also available for computers running the UNIX and MacOS operating systems.

Another interesting program is a free program named *PrintFile*. This little utility program may be set up to utilize drag-and-drop functionality or to send the PostScript file to a local or networked printer. It will even print encapsulated PostScript files. You can find this program at this URL:

<http://hem1.passagen.se/ptlerup/prfile.html>

Producing Encapsulated PostScript Output

To produce PostScript output to be included in other files, such as journal articles or books, you must select the encapsulated option before you send graphics output to a PostScript file:

```
Set_Plot, 'PS'
Device, /Encapsulated
```

IDL encapsulated PostScript has been successfully placed into LaTeX, Microsoft Word, FrameMaker, and other word processing documents.



Note that encapsulated PostScript files will not print on a PostScript printer by themselves, although the printer will churn away. Encapsulated PostScript files lack the PostScript *showpage* command, which is used to eject the page from the printer. The files must be included or "encapsulated" within another document to print properly.

Note also that when you import the encapsulated file into the other document, you probably will not be able to see the graphic until it is printed, unless you have a PostScript previewer or you use the *Preview* keyword described below.

To turn encapsulated PostScript off, set the *Encapsulated* keyword to 0, like this:

```
Device, Encapsulated=0
```

Encapsulated PostScript Graphic Preview

In the normal case, encapsulated PostScript files are not displayed in the document into which they have been imported. That is to say, the frame that holds the imported file usually is blank or is grayed out. However, the PostScript graphic will print properly when the document is sent to a PostScript printer.

If you would like to be able to see a representation of the graphic in the document into which you imported the file, you must specify a value for the *Preview* keyword. A *Preview* keyword value of 1 will cause the PostScript driver to include a bit-mapped image of the graphic along with its PostScript description. The bit-mapped image will be shown in the document frame and the PostScript description will be used when the document is printed.

```
Device, /Encapsulated, Preview=1
```



Note that not all word processing software is capable of displaying a bitmapped preview image. For example, I've found that a preview image doesn't display very well in either Microsoft Word or Framemaker on a Macintosh. But the preview image is helpful using Framemaker on a Windows machine. Experiment with your word processing software to see how well it displays on your machine.

To turn the preview option off, set the *Preview* keyword to 0, like this:

```
Device, Preview=0
```



Note that in IDL 5.2 the preview option was improved dramatically for Macintosh and Windows computers. By setting the *Preview* keyword to 2, you can create a PostScript encapsulated interchange file that has a TIFF image for preview display. The PostScript portion of the file is used for printing on a PostScript printer. While this option works reasonably well for image output, I find it poor for general graphics output. I prefer to open my encapsulated PostScript output created in IDL without a preview option in a program like *GhostScript* or *GhostView*, and use the “add preview” options of these programs to add a preview image. These programs are better equipped to produce a preview image that looks like the final PostScript output than IDL is.

Producing Color PostScript Output

Color PostScript output is supported in IDL. To enable color output, use the *Color* keyword with the PostScript device:

```
Set_Plot, 'PS'  
Device, Color=1
```

Setting the *Color* keyword automatically copies the current color table into the PostScript file. (Similar to the *Copy* keyword to the *Set_Plot* command below.) Be aware that the PostScript device almost always supports 256 colors, which is usually more than you are using on your display device. This can affect your output. See “Problem: PostScript Devices Often Have More Colors Than the Display Device” on page 191 for more information.

Another way to load the color table automatically is to use the *Copy* keyword of the *Set_Plot* command when you set the graphics device to PostScript:

```
IDL> Set_Plot, 'PS', /Copy
```

This command automatically loads the current color vectors into the PostScript file as the first operation it performs when it opens the file. Note that it is the display color tables that are copied into the PostScript files. Often these display color tables do not have as many colors as the PostScript color tables. See “Problem: PostScript Devices Use Background and Plotting Colors Differently” on page 189 for more information.

Once you have made the PostScript device the current graphics device, you can load color tables using the normal color table loading commands. For example, you can type commands like this:

```
IDL> LoadCT, 5, NColors=200
IDL> TVLCT, [70, 255], [70, 255], [70, 0], 200
```

To turn the color option off, set the *Color* keyword to 0, like this:

```
Device, Color=0
```



Note that if your graphics commands use color indices other than the color index specified by *!P.Color* that you *must* set the *Color* keyword to 1. If you fail to do this it is quite likely that your graphics will not show up at all in the PostScript output. Since I almost always write programs that use color indices, I set the *Color* keyword to 1 as a matter of course for all PostScript output. It is a good habit to get into. If you send the output to a gray-scale printer, the colored output will be displayed in dithered gray lines. This is *much* better than no colors at all, believe me.

Color and Gray Scale Images in PostScript

By default, the PostScript device saves only four bits of information per image pixel. This is enough information for only 16 colors or shades of gray. If you want more colors in your PostScript output, the *Bits_Per_Pixel* keyword to the *Device* command should be set to eight. For example, to output an image using the full 256 colors that the PostScript device is capable of producing, you might configure the device like this:

```
IDL> image = LoadData(7)
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PS'
IDL> LoadCT, 33
IDL> Device, Color=1, Bits_Per_Pixel=8, Filename='test.ps'
IDL> TV, image
IDL> Device, /Close_File
IDL> Set_Plot, thisDevice
```

If you have a PostScript previewer available, open the file *test.ps* and see what the output looks like.

True-Color Images

The PostScript device can also display a 24-bit color or true-color image. A true-color image is a 3D array, with one dimension set to 3. For example, an *m*-by-*n* true-color image can be pixel interleaved (3, *m*, *n*), row interleaved (*m*, 3, *n*), or image interleaved (*m*, *n*, 3).

The PostScript device is equivalent to an 8-bit display device. This means you have the same issues to consider when you display a 24-bit image as you do for any other 8-bit device. (See “Displaying 24-Bit Images” on page 64 for additional information.) For example, here is how a pixel interleaved, true-color image might be sent to a PostScript device as a color image:

```
IDL> rose = LoadData(16)
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PS'
IDL> Device, Color=1, Bits_Per_Pixel=8
IDL> image2d = Color_Quan(rose, 1, r, g, b)
IDL> TVLCT, r, g, b
IDL> TV, image2d
IDL> Device, /Close_File
IDL> Set_Plot, thisDevice
```



Note that the *TVImage* command that you downloaded to use with this book automatically knows if image output is to the PostScript device and sets the image up appropriately for viewing. All that is required if you are using *TVImage* are these commands:

```
IDL> rose = LoadData(16)
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PS'
IDL> Device, Color=1, Bits_Per_Pixel=8
IDL> TVImage, rose
IDL> Device, /Close_File
IDL> Set_Plot, thisDevice
```

Before you move on to the rest of this chapter, be sure your current graphics output device is your display device. If you are unsure, type the following command, which will give you details of how your current graphics device is configured:

```
IDL> Help, /Device
```

If your display device is not the current graphics device, make it so by typing the command below. Use the display designation that is appropriate for your machine.

```
IDL> Set_Plot, 'X' ; or 'Win' or 'Mac'
```

Creating Quality Output on PostScript Devices

The secret to creating quality hardcopy output that looks similar to—if not exactly like—the output on your display device is to understand the similarities and differences of your display and your output device. For example, consider the similarities between your display device and the PostScript device.

Similarities Between the Display and PostScript Devices

The most obvious similarity is that you create a graphic window on each device for displaying your data, although the way you do it is different on each device. On your regular display device you might create a graphic window by typing this:

```
Window, XSize=300, YSize=400, XPos=100, YPos=200
```

You have created a graphic window that has a size of 300 pixels in the X direction and 400 pixels in the Y direction. You located it on your display (which might, for example, have a size of 1024-by-768 pixels) with its lower-left corner at 100 pixels in X and 200 pixels in Y.

You do a similar thing when you create a graphics window in PostScript. The difference is that you don't use the *Window* command to create the window. (The *Window* command is an invalid command when the PostScript device is the current graphics output device, a point you should keep in mind as you write your IDL programs.) Instead, you use the *Device* command to tell the PostScript device how big its graphic window should be. Like this:

```
Device, XSize=3, YSize=4, XOffset=1, YOffset=2, /Inches
```

Thinking this way, you see that the PostScript page is analogous to your entire display device and the area described by the *Device* command is analogous to the graphics window on your display. In other words, what you do by issuing commands like the two above is to create a location on the display or page where you are going to draw your graphics output.

IDL uses its normal rules to put graphics into either window. So what happens when you issue a command like this?

```
Plot, FIndGen(11)
```

IDL uses its normal rules to position the graphic into the window. In this case, IDL calculates the size of a character in device units and uses this to determine default margins for the plot. The plot will go into the window based on those margins and will, in general, be positioned to fill up the graphics window.

But will what you see on your display be the same as what you see in your PostScript output? Probably not, although it should be similar. The reason it is not exactly the same is explained by the ways in which your display device is different from your PostScript device.

Differences Between the Display and PostScript Devices

There are several differences between your display device and your PostScript device that are critical for you to understand if you want to produce PostScript output nearly identical to what you see on the display. With one or two exceptions, these differences are not particularly large, nor do they seem particularly important, but it has been my experience that if you fail to understand the differences, you will be extremely frustrated in your efforts to produce quality hardcopy output.

Problem: PostScript Windows May Have a Different Aspect Ratio

The first, relatively minor difference is that the graphics window you create on your display device may not have the same aspect ratio (ratio of height to width) as the graphics window you create on your PostScript page. This is not too surprising, since the windows are created in different ways: with the *Window* command on your display, and with the *Device* command in PostScript.

In fact, most people who display graphics in IDL don't even open windows with the *Window* command. They simply execute a *Plot* or *Surface* command and a window opens for them. The default window size varies from machine to machine and can often be configured by the user. On workstations, the default window size is usually 640-by-512 pixels. On PCs, the default window size is most often one-quarter of the display size. On my Windows NT machine, it is 512-by-384. In the PostScript device, the default window size is 7-by-5 inches. The aspect ratios (the Y size divided by the X size) for these three possibilities are 0.800, 0.750, and 0.714, respectively.

Clearly, the output of the same *Plot* command will look different in all three windows, since the aspect ratio of all three windows is different and since IDL acts to fill up whatever window space is available with the *Plot* command.

Solution: Make the Aspect Ratios of Graphics Windows the Same Size

So, the first rule of creating virtually identical graphical output is to be sure that your display window and your PostScript window have the same aspect ratio. This is quite simple to do. Just calculate the aspect ratio of your current display window and set your PostScript window accordingly. For example (assuming you have a graphics window open on your display), you might type this:

```
IDL> aspectRatio = Float(!D.Y_VSize)!D.X_VSize
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PS'
IDL> Device, XSize=5, YSize=5*aspectRatio, /Inches
IDL> Plot, LoadData(1)
IDL> Device, /Close_File
IDL> Set_Plot, thisDevice
```

The chances of the plot looking the same in the display window and in your PostScript output are now much higher than they were before.

I like to use the *PSWindow* command to create a PostScript graphics window that has the same aspect ratio as the current display window. (The program *pswindow.pro* is among the files you downloaded to use with this book.) The program returns (in inches, by default) the sizes and offsets necessary to create the largest graphics window possible on the PostScript page with the same aspect ratio as the current graphics window. The return value is used to set the appropriate *Device* command keywords, usually through its *_Extra* keyword. (See “Passing Undefined Keywords by Keyword Inheritance” on page 216 for additional information about the *_Extra* keyword.)

To see how it is used, first open a graphics display window and display a line plot.

```
IDL> Window, XSize=400, YSize=300 ; Aspect Ratio = 0.75  
IDL> curve = LoadData(1)  
IDL> Plot, curve
```

Now create a PostScript window with the same aspect ratio and draw the plot in it.

Type:

```
IDL> rightSize = PSWindow()  
IDL> thisDevice = !D.Name  
IDL> Set_Plot, 'PS'  
IDL> Device, _Extra=rightSize, /Inches, File='test.ps'  
IDL> Plot, curve  
IDL> Device, /Close_File  
IDL> Set_Plot, thisDevice
```

If you have a PostScript printer available or a PostScript previewer, send the file to it. Compare the output with what is in your display window. Is it identical?

No? But almost? Ok, read on.

Problem: PostScript Devices Have a Higher Display Resolution

In the default case, IDL calculates the margins of the plot, which are based on character size, and determines where to place the plot axes in the graphics window. But the character size on which the margins are calculated is not the same in PostScript as it is on your display device.

The reason for this is that the PostScript device has a much finer resolution than your display device. You can examine the IDL system variables *!D.X_PX_CM* (to determine the number of pixels per centimeter) and *!D.X_CH_SIZE* (to determine the X size of the default character in device units) to see what the differences are. Type:

```
IDL> thisDevice = !D.Name  
IDL> Print, !D.X_PX_CM, !D.X_CH_SIZE  
IDL> Set_Plot, 'PS'  
IDL> Print, !D.X_PX_CM, !D.X_CH_SIZE  
IDL> Set_Plot, thisDevice
```

The numbers produced on a Macintosh computer, for example, are these:

```
Mac:    28.35   6  
PS:    1000.00 222
```

In other words, for each pixel on your display screen, there are about 35 pixels in the PostScript device. Moreover, the ratio of character size to resolution is different for the two devices. The ratio is 0.212 for the Macintosh and 0.222 for PostScript.

You can see right away that it is going to be a bad idea to position any kind of a graphic in IDL with device or pixel coordinates, unless this resolution factor is taken into account. For example, suppose you want to draw a box around a particular portion of an image which goes from 100 to 200 pixels in the X direction and from

150 to 250 pixels in the Y direction on your display. You might create and draw the box like this:

```
xBox = [100, 100, 200, 200, 100]
yBox = [150, 250, 250, 150, 150]
PlotS, xBox, yBox, /Device
```

In a 400-by-400 pixel display window, the ratio of box size to window size is going to be 1:16. In a 10-centimeter by 10-centimeter PostScript window (about 4-by-4 inches), the ratio of box size to window size is going to be 1:10,000! This is a pretty small box and almost certainly *not* what you wanted.

Solution: Don't Use Device Coordinates to Position Graphics

So, the second rule of creating virtually identical graphical output is to be sure you position graphical elements in your output window with data or normalized coordinates, and *not* with device coordinates.

For example, the box above will enclose the same relative proportion of both the display window and the PostScript window, if it is defined like this:

```
xBox = [0.250, 0.250, 0.500, 0.500, 0.250]
yBox = [0.375, 0.625, 0.625, 0.375, 0.375]
PlotS, xBox, yBox, /Normal
```

Another way the ratio of character size to resolution affects output is in the way the graphic output is placed in the graphics window. Recall that by default IDL uses margins to place the graphic in the window and that margins are calculated based on character size. If the character size is different on your display and on the PostScript output, this will affect your graphics output slightly.

You can compensate for this by positioning your plots with the *Position* keyword, which uses normalized coordinates to place the axes at an exact location in the output window, whether you are using a display window or a PostScript window. (See “Setting the Graphic Position” on page 46 for more information.)

The simple plot above could be put into either window in the same way by using this command.

```
Plot, LoadData(1), Position=[0.1, 0.1, 0.95, 0.95]
```

Problem: PostScript Devices Can Use Different Display Fonts

IDL, by default, uses Hershey fonts for its graphical output. The Hershey set is an example of what are called *vector fonts*. These fonts are described as a set of vectors and are drawn as such when they are rendered. There are two huge advantages to using vector fonts: they can be scaled and rotated in 3D space easily and they are device independent. The major disadvantage of vector fonts is that on a high-resolution output device like a PostScript printer, they do not have the same high quality as true PostScript fonts.

For this reason, many people prefer to use PostScript fonts when they output their plots to PostScript. This causes the plot on your display to look slightly different from the plot in PostScript, because there is no one-to-one relationship between the Hershey vector fonts and the fonts available on a PostScript printer. PostScript fonts must be substituted for Hershey fonts. This causes the font characters to be different sizes, which can sometimes cause text alignment problems.

Solution: Take Care in Designing and Positioning Text

The solution is to take care how you design your textual output. For example, titles should be centered about points or specifically aligned to points using the *Alignment*

keyword to the *XYOutS* command, if possible. You see an example of the difference between Hershey fonts and true Postscript fonts in Figure 76.

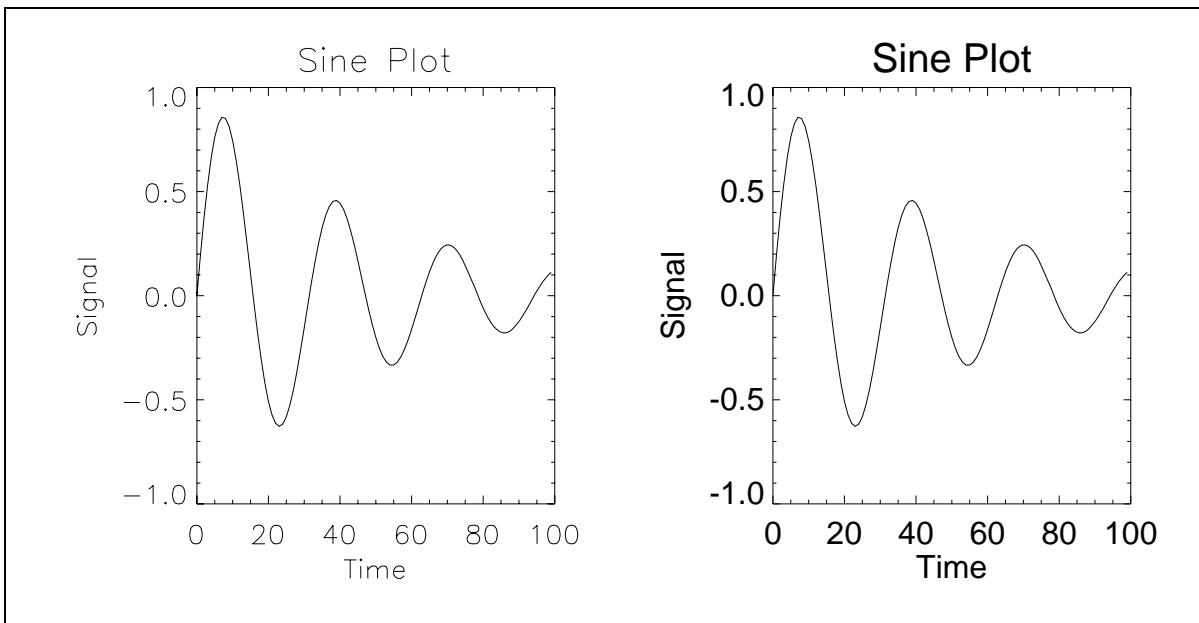


Figure 76: The figure on the left was created with the Hershey Simplex Roman font. The figure on the right was created with PostScript Helvetica font. The two plots look similar, but not the same.

To select a true PostScript font, set the *!P.Font* system variable or *Font* keyword to zero. In the absence of any other information, IDL maps the Hershey fonts to PostScript fonts according to the mapping in Table 14. You can always discover what this mapping is in IDL by typing this:

```
Set_Plot, 'PS'
Help, /Device
```

Notice from the table below that the normal default font, Simplex Roman, is mapped to Helvetica. The Helvetica font is proportionally larger than the Simplex Roman font at the same font size in the PostScript output. This means that you must take care when using font substitution to position text in your graphic output in a way that is reasonably portable from the display to hardcopy.

In practice, this usually means centering or left and right justifying the output text to a normalized coordinate. For example, here is code that will build a simple legend on a plot:

```
PlotS, [0.2, 0.3], [0.7, 0.7], /Normal
PlotS, [0.2, 0.3], [0.6, 0.6], LineStyle=3, /Normal
XYOUTS, 0.32, 0.7, 'Normal bias', /Normal, Alignment=0.0
XYOUTS, 0.32, 0.6, 'No bias', /Normal, Alignment=0.0
```

This code is positioned at the same relative location in both the display window and in the PostScript window.

Note that you can change the font mapping with the *Device* command. For example, font !4 is normally mapped to the Helvetica-Bold PostScript font. If you want to change this to a Palatino-Bold-Italic font, you can do this:

```
Device, /Palatino, /Bold, /Italic, Font_Index=4
```

To use the Palatino-Bold-Italic font in the legend above, you would type:

Number	Hershey Font	PostScript Font
!3	Simplex Roman	Helvetica
!4	Simplex Greek	Helvetica-Bold
!5	Duplex Roman	Helvetica-Narrow
!6	Complex Roman	Helvetica-Narrow-BoldOblique
!7	Complex Greek	Times-Roman
!8	Complex Italic	Times-BoldItalic
!9	Math and Special	Symbol
!10	Special	ZapfDingbats
!11	Gothic English	Courier
!12	Simplex Script	Courier-Oblique
!13	Complex Script	Palatino-Roman
!14	Gothic Italian	Palatino-Italic
!15	Gothic German	Palatino-Bold
!16	Cyrillic	Palatino-BoldItalic
!17	Triplex Roman	AvantGarde-Book
!18	Triplex Italic	NewCenturySchlbk-Roman
!19	Undefined	NewCenturySchlbk-Bold
!20	Miscellaneous	Undefined

Table 14: *The default mappings from Hershey vector fonts, which are normally used on the display, and the PostScript fonts that are used in PostScript output.*

```

Plots, [0.2, 0.3], [0.7, 0.7], /Normal, Font=0
Plots, [0.2, 0.3], [0.6, 0.6], LineStyle=3, /Normal
XYOUTS, 0.32, 0.7, '!4Normal bias!X', /Normal, Align=0.0
XYOUTS, 0.32, 0.6, '!4No bias!X', /Normal, Align=0.0

```

The `!X` in the strings above causes the default font to revert back to whatever font was in place before it was switched to the `!4` font. See “Adding Text to Graphical Displays” on page 51 for more information about using the `XYOutS` command.

Problem: PostScript Devices Use Background and Plotting Colors Differently

Another difference between your display device and the PostScript device is that PostScript devices work with colors differently. For example, the PostScript device reverses the normal background and plotting colors. For output on your display, the background color is controlled by the `!P.Background` system variable. When you start IDL, this system variable is set to the value 0, meaning IDL uses the 0th index in the current color table as the background color. In most of the color tables supplied with IDL this is a black color.

But the PostScript device essentially ignores any background color information and always renders the background in white. (Or, more accurately, the PostScript device doesn't draw a background at all. The color of the paper in the printer is the background color.) For example, you might try to type these commands to get a green plot on a charcoal background:

```
TVLCT, [0, 70], [255, 70], [0, 70], 100
Plot, LoadData(1), Color=100, Background=101
```

While these two commands render the output correctly on the display, they render a green plot on a white background in PostScript. This is essentially because filling the page with a background color is a raster operation (i.e., individual pixels are rendered), whereas most direct graphics commands in IDL are vector operations. To get the PostScript device to perform raster operations, you have to issue a raster command such as *TV* or *Polyfill*, as described below.

Solution: Understand How PostScript Handles Background and Plotting Colors

The only way to get a different color background in PostScript output is to essentially render the background as an image of a particular color. For example, you can use the *PolyFill* command like this:

```
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PS'
IDL> Device, /Color, Bits_per_Pixel=8, File='example.ps'
IDL> TVLCT, [0, 70], [255, 70], [0, 70], 100
IDL> Polyfill, [0,1,1,0,0], [0,0,1,1,0], /Normal, Color=101
IDL> Plot, LoadData(1), Color=100, /NoErase
IDL> Device, /Close_File
IDL> Set_Plot, thisDevice
```

The plotting color is also changed by the PostScript device. The *!P.Color* system variable is normally set to the color index number of the last color in your color table (if you are on an 8-bit display) or to the number 16777216, which can be decomposed into a white color (if you are on a 24-bit display). For example, if you have 220 colors in your 8-bit IDL session, *!P.Color* will normally be set to 219. When you select the PostScript device, the *!P.Color* system variable is always set to 0.

What this means to you is that if you draw a plot on the display with IDL's default gray-scale color table loaded, you will see a white plot on a black background. If you do the same plot into a PostScript file, you will see a black plot on a white background. You see this illustrated in Figure 77.

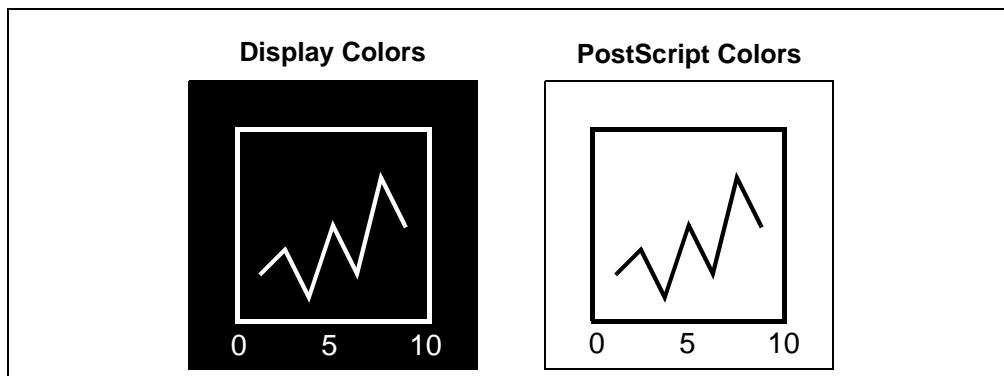


Figure 77: PostScript output reverses the background and plotting colors for most plots, so that output that appears white on black on your display, is black on white in PostScript.

The plotting color, unlike the background color, is always honored by the PostScript device. Thus, you can set `!P.Color` to point to a color index other than 0 and you will render your plots in that color, both on the display and in your PostScript output. You can also render plots with the `Color` keyword and the color specified will be honored both on the display and in PostScript, but *only* if the `Color` keyword is set for the PostScript device. For example, these two commands always draw the plot in red, whether on your display or in PostScript output if you have turned color PostScript on:

```
IDL> TVLCT, 255, 0, 0, 100
IDL> Plot, LoadData(11), Color=100
```



Note that on gray-scale printers, colors are represented as dithered lines, and so show up as dotted or broken lines, depending on the resolution of your printer and the “color” that is being represented. In practice, this often means that if you plan to print your output on a gray-scale printer you will want to make sure that your drawing color is black. Also note that if you want gray-scale output you must set the `Color` keyword. If you don’t, you will always get just strict black and white output, no matter what “color” you are trying to represent.

Problem: PostScript Devices Often Have More Colors Than the Display Device

Another way that PostScript devices are usually different from your display device is in the total number of colors they use. PostScript devices are always capable of displaying at least 256 colors. Normally, you use less than 256 colors on your display device if you run on an 8-bit display device. You can tell how many colors you are using in your current IDL session by opening a graphics window and printing the value of the `!D.N_Colors` system variable, like this:

```
IDL> Window
IDL> Print, !D.N_Colors
```

Usually, the number of colors will be in the range of 200 to 240 on an 8-bit display. The value will be “thousands” or “millions” of colors on a 16-bit or 24-bit display. The number of colors will always be less than 256 if you are running IDL on a personal computer with an 8-bit graphics card, and will probably be less on other computers with an 8-bit graphics card unless you have a private color map. The difference can affect how your output looks, if you are not careful in how you display your data.

For example, suppose you have 200 colors in your IDL session and you want to display an image with a gray-scale color table. You might load the color table like this:

```
IDL> LoadCT, 0
```

This command finds the red, green, and blue color vectors that make up the gray-scale color table in the color table file, and re-samples those vectors so they represent the number of colors used in the IDL session. In this case, the vectors that are loaded into the physical color table are 200 elements in length. To display the image, you might type something like this:

```
IDL> image = LoadData(7)
IDL> TVScl, image
```

The image might look like the left-hand image in the illustration in Figure 78.

If you now wanted this image saved to a PostScript file you might try setting the PostScript device to be the current graphics device (with the `Copy` keyword to copy the current color table into the PostScript file), and re-issuing the `TVScl` command above. For example, you might type something like this:

```
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PS', /Copy
IDL> Device, XSize=3, YSize=3, /Inches, /Color, $
```

```
Bits_Per_Pixel=8, File='image.ps'
IDL> TVScl, image
IDL> Device, /Close_File
IDL> Set_Plot, thisDevice
```

But if you did this, you would probably be disappointed in the result. Your output would look something like the right-hand image in the illustration in Figure 78. That is, the shades of gray in your PostScript output will be different from the shades of gray on your display.

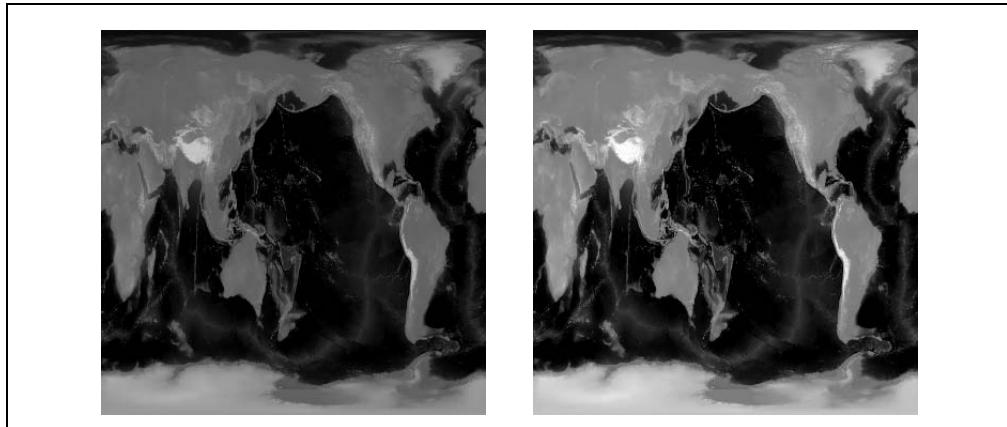


Figure 78: If you are not careful about colors, the output on your display (left illustration) will be different from the output in your PostScript file (right illustration). In particular, those pixels with values greater than the number of colors you have on your display device will be rendered incorrectly. In this case, many of the pixels are rendered too lightly.

The reason for this stems from the way the color tables are loaded into the PostScript device. When you issued the *Set_Plot* command above, IDL copied the colors in the first 200 colors of the display color table into the equivalent colors of the PostScript color table. (This happens whether you use the *Copy* keyword or not.) But it does not affect the colors above color index 200, which had previously been initialized to the gray-scale color table. You can see what is happening by looking at a plot of the red color vector as it appears in the PostScript file. The vector should be linear, but you see in Figure 79, that it has a sharp discontinuity at color index 200. What happens to the image is that any pixel having a value greater than 199 is being displayed in the incorrect color in the PostScript output.

Solution: Be Sure to Scale Your Data Appropriately in PostScript

This problem can be addressed in one of two ways. First, you can reload the color table once you make the PostScript device the current device. Or, you can be sure you scale the image data into the range of colors available on your *display device*. Reloading the color table will make the output on your display and the PostScript output will look *almost* the same. To make the output look identical (within the constraints imposed by the different technologies for producing color, of course), it is necessary to scale the data into the number of colors available on the display device. If the color tables and the data are the identical, the output will also be identical. (See “Scaling Image Data” on page 62 for more information on scaling the data appropriately.)



Note that in the default case only four bits of information is saved for each image pixel in a PostScript image. This means that even if your PostScript device is capable of displaying 256 colors, you will only ever be able to see 16 in your output images. If you

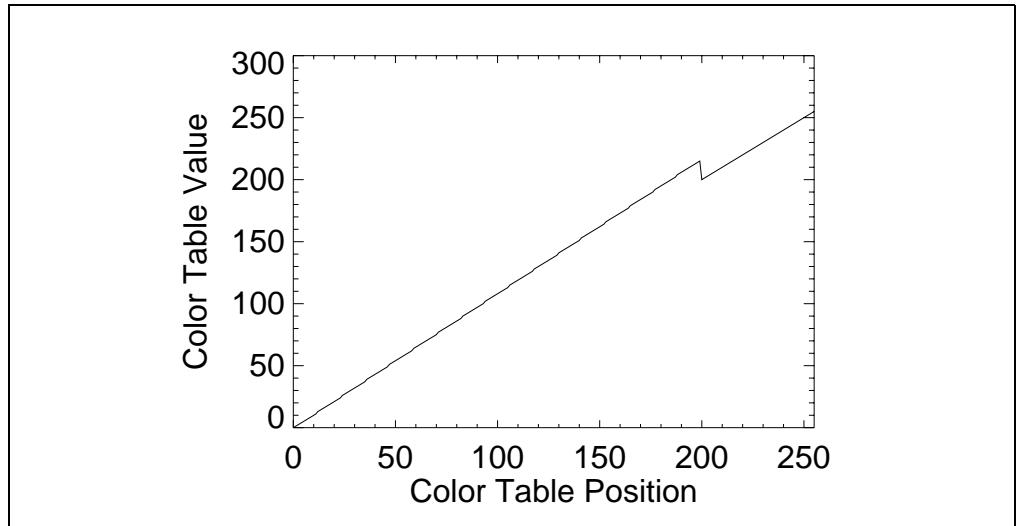


Figure 79: A plot of the red color vector after making the PostScript device the current graphics device. The first 200 colors in the color table were copied over from the display color table. Thus, any pixels having values above 199 will be displayed incorrectly in this PostScript file.

want to see all 256 colors you must store eight bits of pixel information. This is set with the *Bits_Per_Pixel* keyword to the *Device* command, like this:

```
Device, Bits_Per_Pixel=8, Color=1
```

Problem: PostScript Devices Display Images Differently

Another difference between your display device and your PostScript device is that images are displayed differently. In particular, the display device has fixed-sized pixels and the PostScript device has scalable pixels. In other words, in PostScript a single pixel can be virtually *any* rectangular size. This affects the way images are output to the PostScript file.

What PostScript does is to use the size of the PostScript drawing window and the aspect ratio of the image to determine how big an image should be. For example, if the PostScript drawing window is 2-inches-by-2-inches, and the image to be output is 360-by-360 pixels, then a simple *TV* command results in a PostScript image of exactly 2-inches-by-2-inches.

```
IDL> thisDevice = !D.Name
IDL> image = LoadData(7)
IDL> Set_Plot, 'PS'
IDL> Device, XSize=2, YSize=2, /Inches, /Encapsulated
IDL> PlotS, [0, 1, 1, 0, 0], [0, 0, 1, 1, 0], /Normal
IDL> TV, image
IDL> Device, /Close_File
IDL> Set_Plot, thisDevice
```

The commands above produce the output you see in Figure 80.

However, if the output window size does not have the same aspect ratio as the image itself, the image is sized so that it maintains its aspect ratio, and one dimension of the image completely fills the output window. For example, using the same image as above, here is an output box that is size 1 inch in the X direction and 2 inches in the Y direction.

```
IDL> Set_Plot, 'PS'
IDL> Device, XSize=1, YSize=2, /Inches, /Encapsulated
```

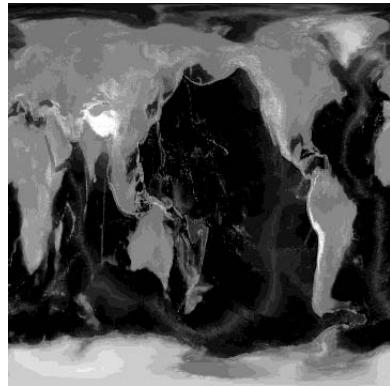


Figure 80: PostScript uses scalable pixels to fit images into the output window size. Here the size is two inches by two inches.

```
IDL> PlotS, [0, 1, 1, 0, 0], [0, 0, 1, 1, 0], /Normal  
IDL> TV, image  
IDL> Device, /Close_File
```

You see the result of these command in Figure 81. Notice that the image is now just a 1-inch-by-1-inch image, and only fills up half the output window.

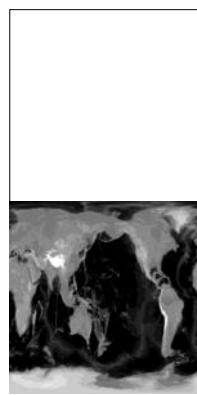


Figure 81: When the output window has a different aspect ratio than the image, the image is sized so that it maintains its aspect ratio and completely fills one dimension of the output window.

Similarly, if you have a 2-inch-by-1-inch output window, like this:

```
IDL> Set_Plot, 'PS'  
IDL> Device, XSize=2, YSize=1, /Inches, /Encapsulated  
IDL> PlotS, [0, 1, 1, 0, 0], [0, 0, 1, 1, 0], /Normal  
IDL> TV, image  
IDL> Device, /Close_File
```

The result can be seen in Figure 82.

If the PostScript drawing window is 1-inch in the X direction and 3-inches in the Y direction, then the *TV* command will result in an image that is 1-inch-by-1-inch.

The fact that images are always sized according to the size of the output window and the aspect ratio of the image can cause difficulties. For example, suppose you had a 500-by-500 pixel display window and you wanted to center an image that was 400-by-400 pixels in the window. Suppose further that you wanted to draw an outline around

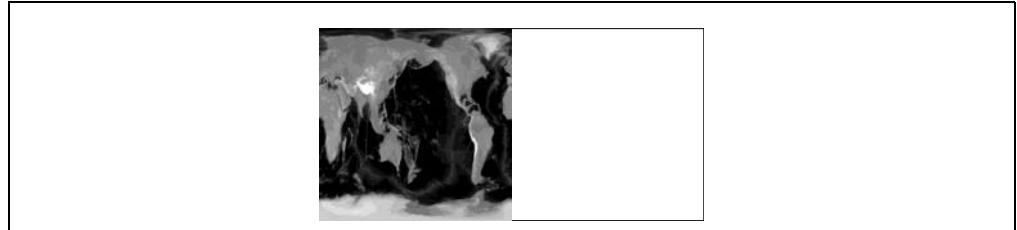


Figure 82: This illustration is similar to Figure 81, except that the output window is twice as big in X direction as it is in the Y direction.

the image. You might use these commands to display and position the image in the window:

```
IDL> image = LoadData(7)
IDL> image = Congrid(image, 400, 400, /Interp)
IDL> Window, XSize=500, YSize=500
IDL> TV, image, 0.1, 0.1, /Normal
IDL> Plot, FIndGen(100), /NoData, /NoErase, $
      Position=[0.1, 0.1, 0.9, 0.9]
```

You can see the output of these commands, issued when your display is the current graphics device, in Figure 83.

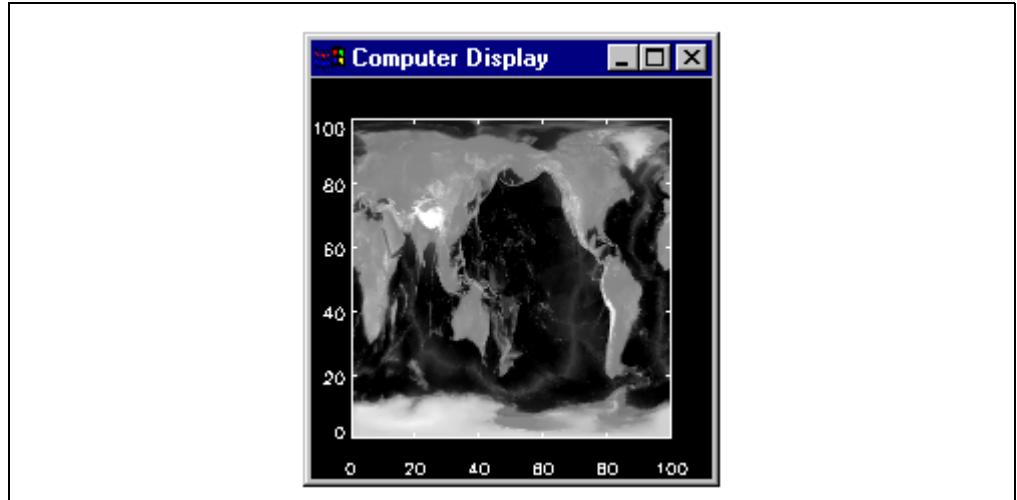


Figure 83: An image with box around it rendered on the display.

But if you issue these same commands (without the *Window* command) in the PostScript device, you may get a much different picture. In particular, the image is sized according to the settings of the output window, which is likely to result in the box not fitting around the image properly, as illustrated in Figure 84

Solution: Size Images with the TV Command

The proper way to size images that are going into PostScript output is to use the *XSize* and *YSize* keywords belonging to the *TV* command. For example, the proper way to get the same kind of output that you saw on the display in Figure 83 is to write the commands like this:

```
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PS'
IDL> Device, XSize=3.5, YSize=2.5, /Inches, /Encapsulated
IDL> TV, image, 0.525, 0.25, XSize=2.8, YSize=2.0, /Inches
```

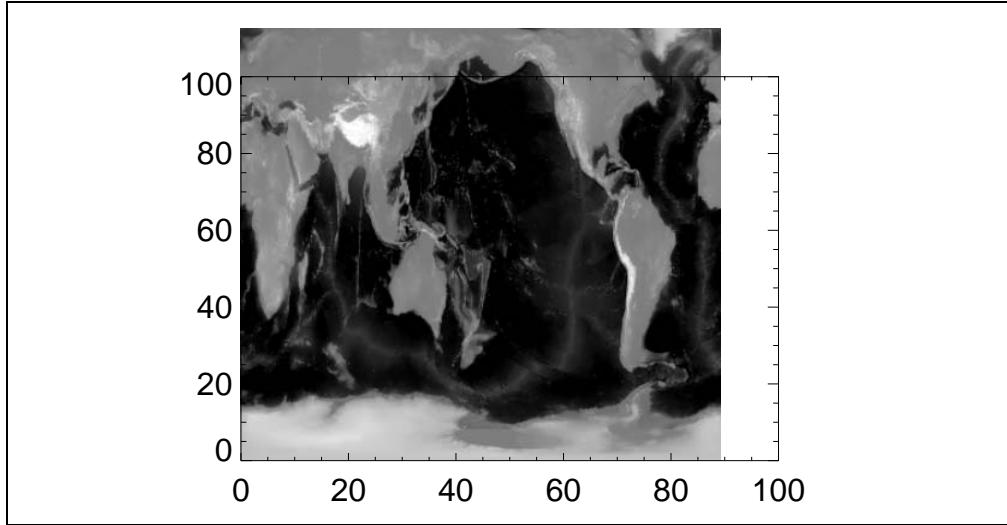


Figure 84: In PostScript the image is sized according to the output window size, which may not be what you want, as illustrated here.

```
IDL> Plot, FIndGen(100), /NoData, /NoErase, $  
      Position=[0.15, 0.10, 0.95, 0.90]  
IDL> Device, /Close_File  
IDL> Set_Plot, thisDevice
```

You see the result of these commands in Figure 85.

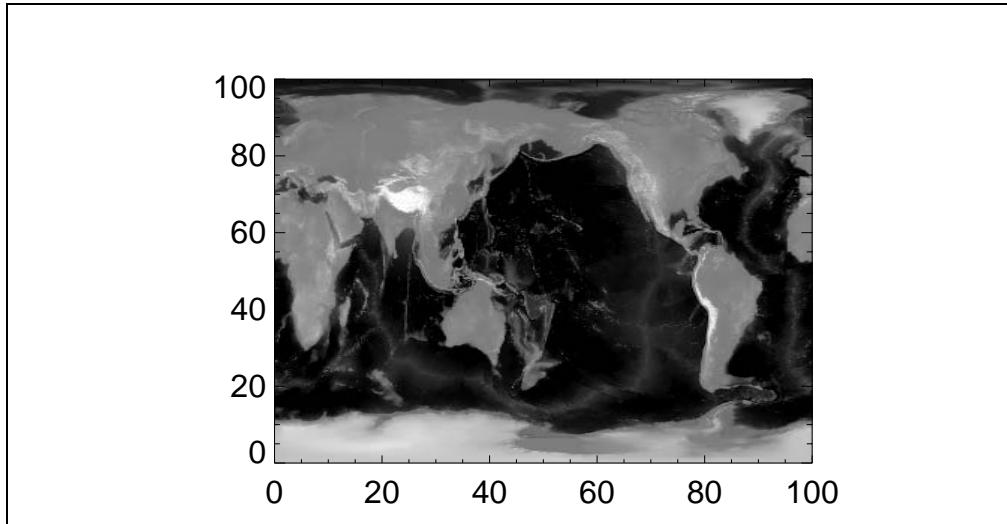


Figure 85: The proper way to size images and place them in a PostScript window is to use the sizing and positioning capabilities of the TV command. Compare this illustration to Figure 84.

If you wanted to write a generic IDL program like the one above that would work no matter what size window it was going into, and would work on your display or in a PostScript file, you might calculate the size of the image and its position in the display window based on device coordinates. The only real difference in working in PostScript and on your display would be how the image is sized. Your program, which might be called *imageax.pro*, might look like this. (This program is among the programs you downloaded to use with this book.)

```
PRO ImageAx, image, Position=position
```

```

IF N_Parms() EQ 0 THEN Message, 'Must pass image argument.'
IF N_Elements(position) EQ 0 THEN $
    position = [0.2, 0.2, 0.8, 0.8]
    ; Get the size of the image in pixel units.
s = Size(image, /Dimensions)
imgXsize = s[0]
imgYsize = s[1]
    ; Calculate the size and starting locations in pixels.
xsize = (position[2] - position[0]) * !D.X_VSize
ysize = (position[3] - position[1]) * !D.Y_VSize
xstart = position[0] * !D.X_VSize
ystart = position[1] * !D.Y_VSize
    ; Size the image differently in PostScript.
IF !D.Name EQ 'PS' THEN $
    TV, image, xstart, ystart, XSize=xsize, YSize=ysize ELSE $
    TV, Congrid(image, xsize, ysize, /Interp), xstart, ystart
    ; Draw the axes around the image.
Plot, FIndGen(100), /NoData, /NoErase, Position=position
END

```

-  Open several windows of various sizes and run this program so that the output is displayed in each of them in turn. Notice that the aspect ratio of the image is not preserved. Rather, it is the position in the window that is preserved.

```

IDL> image=LoadData(9)
IDL> Window, XSize=400, YSize=400, /Free
IDL> ImageAx, image
IDL> Window, XSize=300, YSize=500, /Free
IDL> ImageAx, image
IDL> Window, XSize=600, YSize=300, /Free
IDL> ImageAx, image

```

You can run this program and put the output into any output window, whether on your display or in PostScript. For example, you can send the output to a PostScript file with these commands:

```

IDL> Set_Plot, 'PS'
IDL> Device, XSize=3.5, YSize=2.5, /Inches, /Encapsulated
IDL> ImageAx, image, Position=[0.15, 0.15, 0.95, 0.95]
IDL> Device, /Close_File

```

The result is shown in Figure 86.

To display images in a completely device-independent way, I like to use the *TVImage* program you downloaded to use with this book. Not only does it use the *Position* keyword to position images in the display window in the fashion of *ImageAx* above, it also allows you to maintain the aspect ratio of the image if you like (by setting the *Keep_Aspect_Ratio* keyword). You can learn more about the *TVImage* command in “Positioning Images with Normalized Coordinates” on page 70.

```

IDL> Window, XSize=600, YSize=400, /Free
IDL> TVImage, image, Position=[0.15, 0.15, 0.95, 0.95], $
    /Keep_Aspect_Ratio

```

-  Another extremely important point about displaying images in PostScript is that by default, the PostScript device saves only four bits of information for each image pixel.

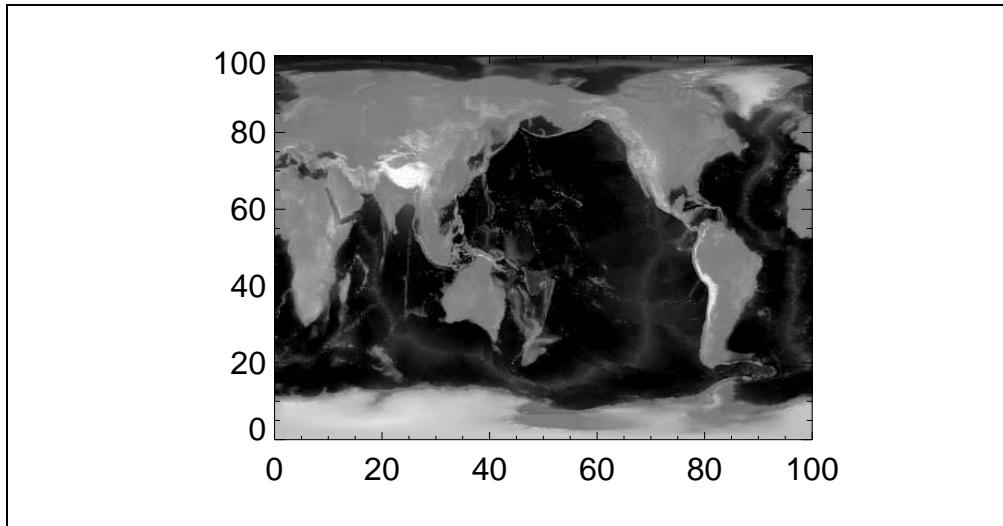


Figure 86: Running the *ImageAx* program into an output window of 3.5 inches by 2.5 inches. Notice the aspect ratio of the image has not been preserved, although its position in the window has been preserved.

This is enough information for only 16 different colors or shades of gray. If you want 256 colors, you must set the *Bits_Per_Pixel* keyword to 8, like this:

```
IDL> Set_Plot, 'PS'  
IDL> Device, Bits_Per_Pixel=8, Color=1
```

Calculating PostScript Offsets in Landscape Mode

The default offset for PostScript files displayed in portrait mode is 0.75 inches in the X direction and 5 inches in the Y direction. This puts the graphic output in the upper half of the page. Thus, it is easy to see that the offsets are calculated from the lower-left corner of the page. (See Figure 87.) You can see the default offsets by typing:

```
IDL> thisDevice = !D.Name  
IDL> Set_Plot, 'ps'  
IDL> Help, /Device  
IDL> Set_Plot, thisDevice
```

When you place the graphics output into landscape mode, however, the entire page is rotated 90 degrees, including the lower-left corner of the page! You can see the default offsets as they actually exist in Figure 87.

If you didn't realize that the offset point actually rotated with the page, you might set the offsets so that the graphic was rotated off the page. For example, suppose you wanted both your X and Y offset set at one inch and you did this:

```
Set_Plot, 'ps'  
Device, XOffset=1.0, YOffset=1.0, /Inches, /Landscape  
Plot, data
```

You can see in Figure 88 the figure you might *think* you are producing, and what you actually produce. Be sure you understand how offsets work in landscape mode.

Configuring the PostScript Device with PSConfig

One of the programs you downloaded to use with this book is a program named *PSConfig*. The purpose of the program is make it possible for a user to interactively

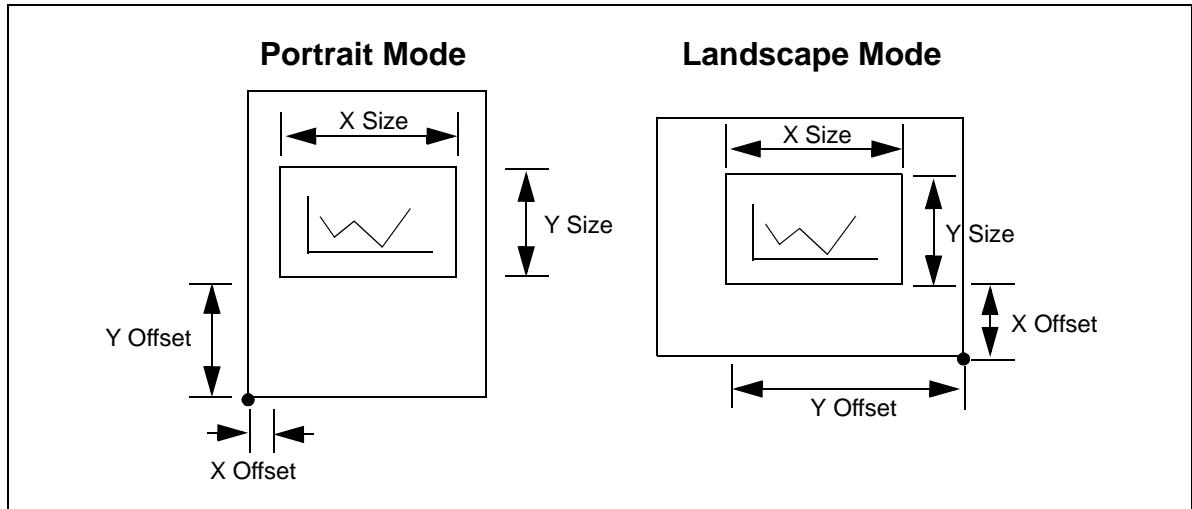


Figure 87: The window sizes and offsets for PostScript portrait and landscape mode. Notice that in landscape mode, the entire page is rotated 90 degrees and that the offsets (but not the window sizes) have rotated with it. Note that this is not true of the Printer device, which always calculates its offsets from the lower-left corner of the page.

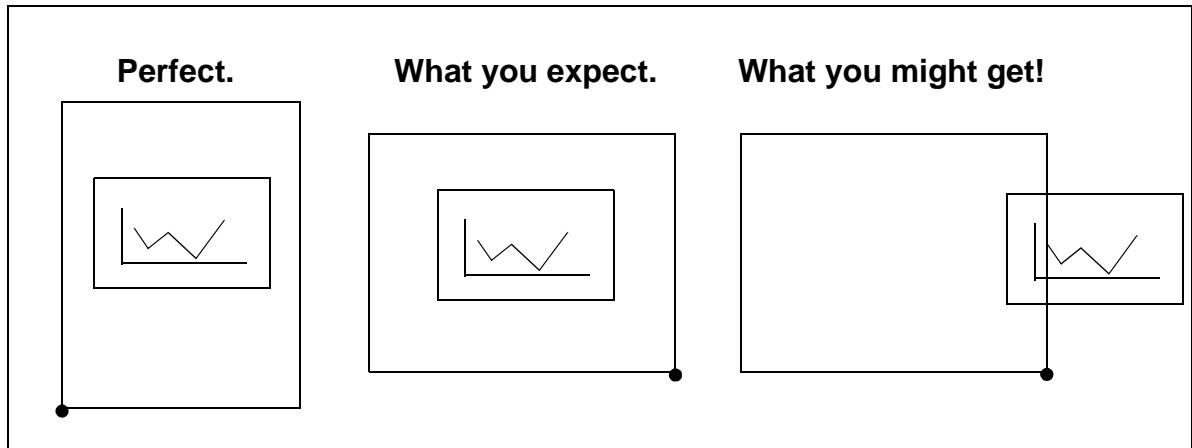


Figure 88: If you don't pay attention to how the landscape offsets work, you may rotate your graphic output right off the page.

determine where on the page the PostScript output should be placed and to set other configurations of the PostScript device. You see an illustration of *PSConfig* in Figure 89. The program is called like this:

```
IDL> deviceKeywords = PSConfig()
```

European users can obtain an A4 page size by using the *European* keyword, like this:

```
IDL> deviceKeywords = PSConfig(/European)
```

The draw widget on the right-hand side represents the PostScript page. The small plot window inside the draw widget delineates the position of the output on the PostScript page. Notice that the plot window can be resized with the mouse as well as dragged around in the window. To center the plot window, click the middle mouse button inside the draw widget window. Pull-down menus provide access to many of the PostScript device configuration parameters. You can specify a file name in the center of the dialog.

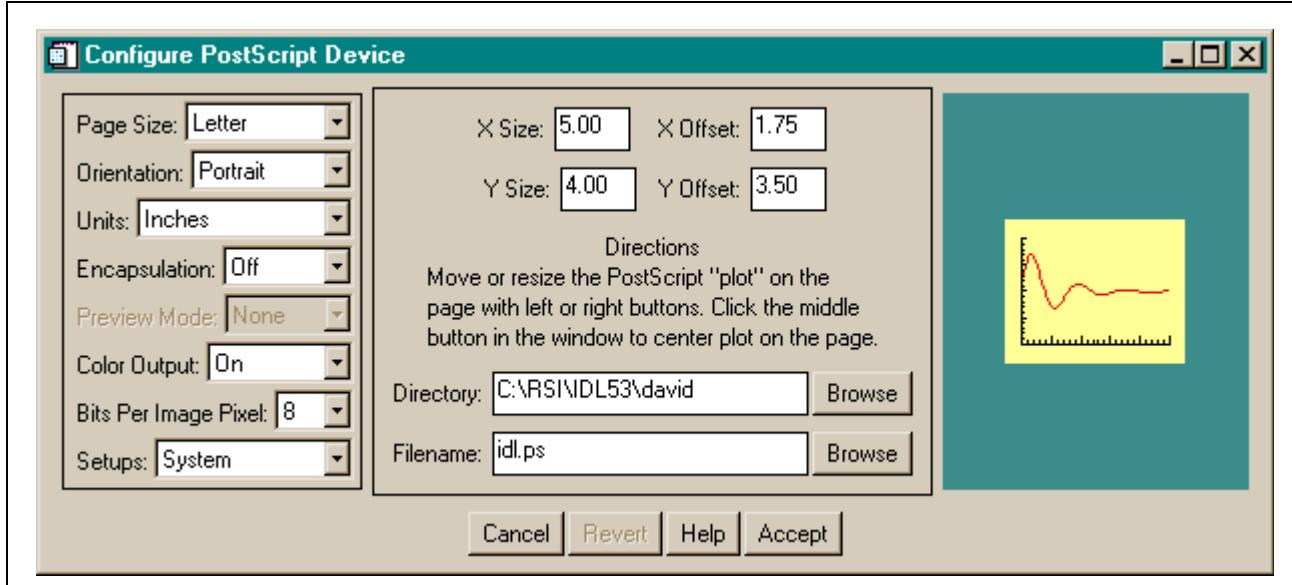


Figure 89: The dialog widget program *PSConfig*. This program provides an interactive way for the user to provide input to how the PostScript device is configured.

When you have things the way you like them, click the *Accept* button. *PSConfig* returns a structure, the fields of which are valid *Device* command keywords. Here is the way *PSConfig* can be used to configure the PostScript device and draw a simple line plot:

```
deviceKeywords = PSConfig(Cancel=canceled)
IF NOT canceled THEN BEGIN
    currentDevice = !D.Name
    Set_Plot, 'PS'
    Device, _Extra=deviceKeywords
    Plot, LoadData(1)
    Device, /Close_File
    Set_Plot, currentDevice
ENDIF
```



Notice that one nice feature of *PSConfig* is that the user is shielded from the rotation of the offset point when the device is set to landscape orientation. To the user it appears as if the offset is always calculated from the lower-left-hand corner.

To see how *PSConfig* can be used, try calling the program *XWindow*, which you downloaded for use with this book. *XWindow* is a “smart” graphics window that can resize itself, load color tables that apply only to it, and send its output to a PostScript file. Try calling it like this:

```
IDL> XWindow, 'Shade_Surf', LoadData(2), /Output, /XColors
```

Try using the *XWindow* program to produce a PostScript file of the window contents. Much of the rest of the book is devoted to a discussion of how to write a program like *XWindow*.

Configuring and Using the Printer Device

The *Printer* device was introduced in IDL 5.0 and was initially not configurable with the *Device* command like other graphics output devices. The *Dialog_PrinterSetup*

command was used instead to access the machine-specific printer configuration dialog for the default printer for your machine. Explaining how default printers are set up and configured for each computer platform is well beyond the scope of this discussion, but generally printer configuration dialogs give you more options for configuring the printer itself and fewer options for positioning the graphical output than, say, you have with the PostScript device.

To give the user more options for easily positioning graphics with the *Printer* device, Research Systems introduced *Device* keywords for the *Printer* device in IDL 5.1.1. (Note that these keywords apply *only* when you are sending direct graphics commands to the printer.) These keywords—*XSize*, *YSize*, *XOffset*, and *YOffset*—work much like the same keywords in other hardcopy output devices, although not exactly. Some of the differences will be pointed out below.

To access the printer configuration dialog for your default printer, type this:

```
IDL> ok = Dialog_PrinterSetup()
```

The dialog on a Windows machine looks like the illustration in Figure 90.

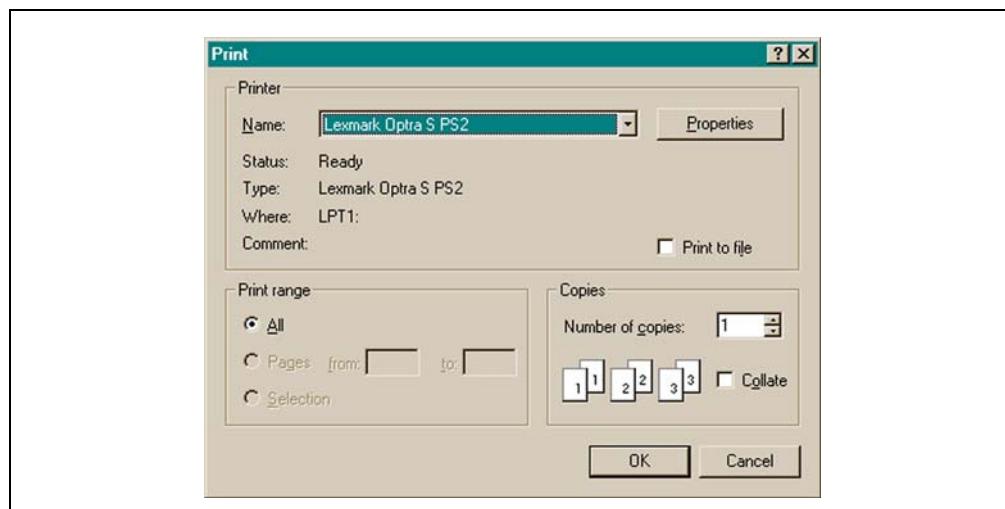


Figure 90: The *Dialog_PrinterSetup* dialog on a Windows NT machine.

The important thing to know about using the *Printer* device is that output is only ejected from the printer when the *Close_Document* keyword is used with the *Device* command. The correct sequence of commands to produce a line plot, for example, will look like the code below. Closing the printer document is essential. If you forget to do this, you will not get output out of your printer. Since this code has an *IF* loop, the code below should be entered into a text editor just as you see it as a main-level IDL program. Save the file as *sendprinter.pro*. You can find this program among the files you downloaded to use with this book.

```
data = LoadData(1)
ok = Dialog_PrinterSetup()
IF ok THEN BEGIN
    thisDevice = !D.Name
    Set_Plot, 'PRINTER'
    Plot, data
    Device, /Close_Document
    Set_Plot, thisDevice
ENDIF
END
```

To run this main-level program and send the output to the default printer attached to your machine, type:

```
IDL> .Run SendPrinter
```

Positioning Graphics with the Printer Device

In the first version of the *Printer* device, when you sent graphics to the default printer the graphics output normally filled the entire page and often looked nothing at all like what you saw on your display. In fact, you had little or no control over where the output should be positioned on the display. Images, for example, were printed in device resolution with their lower-left corner located at the bottom left corner of the page. A 256 by 256 image printed on a PostScript printer with 600 dpi pixel resolution would often appear as a half-inch square unless the proper scaling factor was applied. Pixels were not scaled onto the page as they were with the PostScript device. Neither did the *XSize* and *YSize* keywords to the *TV* or *TVScl* commands apply as they do with the PostScript device. (See, for example, “*Changing Image Size in PostScript*” on page 69.)

This behavior changed in IDL 5.1.1 with the addition of keywords to the *Device* command that could be used with the *Printer* device to position and scale graphical output on the page. Like the same keywords for the PostScript device, the *Printer* device keywords are expressed in centimeters by default. (They can also be expressed in inches if the proper *Inches* keyword is used.)

The default values for the *XSize*, *YSize*, *XOffset*, and *YOffset* keywords are expressed in inches and for output in portrait mode they are as follows:

```
XOffset: 0.75 inches  
YOffset: 5.0 inches  
XSize: 7.0 inches  
YSize: 5.0 inches
```

In landscape mode the default values are:

```
XOffset: 0.75 inches  
YOffset: 0.75 inches  
XSize: 9.5 inches  
YSize: 7.0 inches
```

You see immediately that these default values have been chosen to give the same relatively-sized output on the *Printer* device as they do for the PostScript device. But notice too that the offsets are always computed from the lower-left corner of the page. This is certainly intuitive, but is *not* the way PostScript offsets are computed in landscape mode. (See “*Calculating PostScript Offsets in Landscape Mode*” on page 198.) This means that you will have to be extra careful with landscape offsets if you are trying to write a program that can both create a PostScript file *and* send its graphical display to the printer directly.

To help you calculate the proper values for these keywords, Research Systems also introduced the *Get_Page_Size* keyword for the *Device* command, which can be used to return a two-element vector containing the X size and Y size of the *Printer* device “page”. Oddly, you can only get the page size in device coordinates, even though the page and offset keywords are expressed in inches and centimeters. Thus, to get output exactly where you want it on the page, you may have to make a few calculations

For example, if you wanted the graphical output of a plot command to take up 80 percent of the page, you might type these commands:

```
IDL> thisDevice = !D.Name  
IDL> Set_Plot, 'PRINTER'
```

```

IDL> Device, Get_Page_Size=myPage
IDL> Device, XSize=myPage[0]*0.8, XOffset=myPage[0]*0.1, $
      YSize=myPage[1]*0.8, YOffset=myPage[1]*0.1, /Device
IDL> Plot, Findgen(11)
IDL> Device, /Close_Document
IDL> Set_Plot, thisDevice

```

Rather than do all this fooling around, I prefer to use the program *PSWindow*, which is among the programs you downloaded to use with this book. It will calculate the correct sizes and offsets to center a window on the printer page that has the same aspect ratio as the current graphics window. Be sure to set the *Printer* keyword as well as the *Landscape* keyword if you are sending output to the printer in landscape mode instead of to a PostScript file. Note, too, that you almost always have to set the *Landscape* keyword on the *Device* command before you set the size and offset keywords. I don't know why. But setting them at the same time does not result in the proper margins on many printers.

```

IDL> keywords = PSWindow(/Printer, /Landscape)
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PRINTER'
IDL> Device, Landscape=1
IDL> Device, _Extra=keywords
IDL> Plot, Findgen(11)
IDL> Device, /Close_Document
IDL> Set_Plot, thisDevice

```



Note that not all printers calculate the offsets from the lower-left corner of the page. Some printers calculate offsets from the lower-left corner of the *printable area* on the page. And, sadly, each printer can be different in where this spot is located. You may have to experiment a bit. On my Lexmark PostScript printer, the offset spot is 0.25 inches in from both the left and bottom edge of the paper. So, to get perfectly centered output, I have to subtract 0.25 inches from the offset values I *think* I want. For example, my code might look like this if I am using the offsets calculated in *PSWindow*:

```

IDL> keywords = PSWindow(/Printer)
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PRINTER'
IDL> keywords.xoffset = keywords.xoffset - 0.25
IDL> keywords.yoffset = keywords.yoffset - 0.25
IDL> Device, _Extra=keywords
IDL> Plot, Findgen(11)
IDL> Device, /Close_Document
IDL> Set_Plot, thisDevice

```

Another way to subtract the printable area offsets is to use the *[XY]Fudge* keywords to the *PSWindow* command. The fudge units are the same units as used for the size and offset values. If both the X and Y printable area offsets are the same, you can set them both with the *Fudge* keyword, like this:

```

IDL> keywords = PSWindow(/Printer, Fudge=0.25)
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PRINTER'
IDL> Device, _Extra=keywords
IDL> Plot, Findgen(11)
IDL> Device, /Close_Document
IDL> Set_Plot, thisDevice

```

Outputting Images with the Printer Device

Outputting an image to the *Printer* device is also slightly different from outputting the image to a PostScript file with the *PS* device. (See "Problem: PostScript Devices

“Display Images Differently” on page 193.) The main difference is that image aspect ratios are not preserved by the *Printer* device like they are for the PostScript device. But, like the PostScript device, the *Printer* device expects you to use the *XSize* and *YSize* keywords on the *TV* or *TVScl* commands to size the image appropriately. The image offsets can be set with the *XOffset* and *YOffset* keywords to the device command.

For example, suppose you will to place a normally square image in the center of the page with an aspect ratio on the page of 2/3. You might type something like this:

```
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PRINTER'
IDL> Device, XOffset=1.25, YOffset=3.5, /Inches
IDL> image = LoadData(7)
IDL> TV, image, XSize=6, YSize=4, /Inches
IDL> Device, /Close_Document
IDL> Set_Plot, thisDevice
```

This problem too is taken care of by the *TVImage* command, which will simply fill up the window created with the *PSWindow* program with the image. (Or, you can use the *Position* keyword with *TVImage* to position the image in the window in the normal fashion.)

```
IDL> keywords = PSWindow(/Printer, /Landscape)
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PRINTER'
IDL> Device, _Extra=keywords
IDL> TVImage, LoadData(7)
IDL> Device, /Close_Document
IDL> Set_Plot, thisDevice
```

Another huge advantage of the *TVImage* command is that it will work equally well with 8-bit and 24-bit images when outputting to the *Printer*, which is an 8-bit device.

Loading Colors in the Printer Device

Through version 5.3 of IDL (at least) there has been a strange bug in the *Printer* device when outputting to a PostScript printer. The bug is seen if you load a single color with the *TVLCT* command while in the *Printer* device. No matter what color index you use for the color, all subsequent graphics commands that use color index 0 will also be rendered in that color.

This problem is most often seen when you combine graphics commands with image displays. Here is code that demonstrates the problem. Image colors are loaded outside the *Printer* device, and the color table is copied to the *Printer* device with the *Copy* keyword on the *Set_Plot* command. Then, after switching to the *Printer* device, a single yellow color is loaded with *TVLCT* in an index that is not used by the image. Subsequent display of the image results in all pixels having the value 0 being displayed in yellow. The results are shown in Figure 91.

```
IDL> LoadCT, 0, NColors=150 ; Load image colors.
IDL> image = BytScl(LoadData(7), Top=149)
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PRINTER', /Copy
IDL> TVLCT, 255, 255, 0, 200 ; Load yellow at index 200.
IDL> TVImage, image
IDL> Device, /Close_Document
IDL> Set_Plot, thisDevice
```

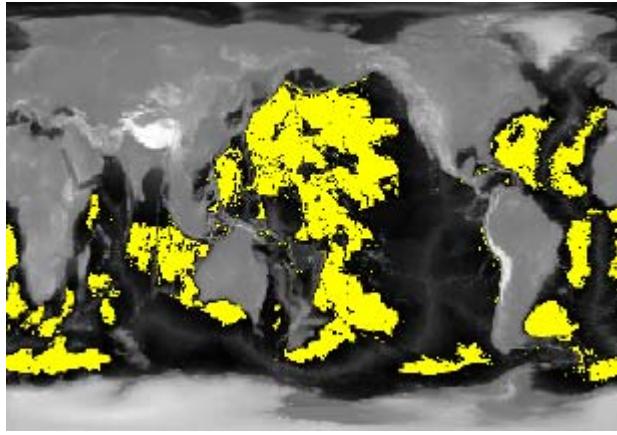


Figure 91: A demonstration of a *Printer* bug that occurs when a single color is loaded while in the *Printer* device.

Color Loading Work-Arounds

There are two ways to work around this bug. The easiest way is to simply load all colors outside the *Printer* device and copy the color table to the *Printer* device with the *Copy* keyword to the *Set_Plot* command. The code above might look like this:

```
IDL> LoadCT, 0, NColors=150 ; Load image colors.
IDL> TVLCT, 255, 255, 0, 200 ; Load yellow at index 200.
IDL> image = BytScl(LoadData(7), Top=149)
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PRINTER', /Copy
IDL> TVImage, image
IDL> Device, /Close_Document
IDL> Set_Plot, thisDevice
```

Sometimes this is not possible. For example, if you are writing a device-independent graphics display routine, your program may not know (or care, hopefully) if it is executing on the display device or the *Printer* device. In that case, you have to be careful to re-display the color table after loading a single color with *TVLCT*. It appears that the color index 0 color doesn't really get *loaded* in the color table. It just *acts* as if it is loaded. For example, if this code were in a device-independent graphics display routine, it might be written like this:

```
LoadCT, 0, NColors=150 ; Load image colors.
TVLCT, 255, 255, 0, 200 ; Load yellow at index 200.

; Reload color table for PRINTER device.

TVLCT, r, g, b, /Get
TVLCT, r, g, b

image = BytScl(LoadData(7), Top=149)
TVImage, image
```

Chapter 8

◆ Discovering the Possibilities ◆◆◆



IDL Programming Fundamentals

Chapter Overview

The purpose of this chapter is to learn the fundamentals of IDL programming. Specifically you will learn:

- The difference between an IDL batch file, main-level program, procedure and function
- How to pass information into and out of IDL programs
- How to use positional and keyword parameters in IDL programs
- How to compile and run IDL programs
- The syntax of many common programming control statements

If you think of an IDL program as a sequence of IDL commands in a file, then there are four types of IDL programs—called IDL program modules—you can write: (1) a batch file, (2) a main-level IDL program, (3) an IDL procedure, and (4) an IDL function.

Writing an IDL Batch File

The simplest program module is an IDL batch file. A batch file consists of a sequence of IDL commands that are identical to the IDL commands you might type at the IDL command line. Most people use a batch file to automate a small set of commands they find themselves typing over and over again at the IDL command line.

For example, suppose you wanted to open and display a number of image files in IDL. If you had read the image data into a variable named *image*, the sequence of commands you might use to display this image data might look like this:

```
IDL> thisImage = BytScl(image, Top=199)
IDL> LoadCT, 5, NColors=200
IDL> s = Size(image, /Dimensions)
IDL> Window, /Free, XSize=s[0], YSize=s[1]
IDL> TV, thisImage
```

These five lines of code are not onerous, but after typing them three or four times, you might decide to put them in a text file named *imageout.pro*. This type of file is known as a *batch file*.

To execute the commands in the file, you place an @ character as the first character on the IDL command line and follow it with the name of the file. (The *.pro* extension is assumed. But if you have given the file some other kind of file extension, you will have to include it with the file name.) For example, like this:

```
IDL> @imageout
```



Notice that the file name is not in quotes. This is different from the normal protocol with file names in IDL.

IDL interprets the commands in the batch file *exactly* as if you were typing the commands at the IDL command line. This means that you might need to use line continuation characters (\$) and other command line syntax to make the command identical to the command you would type at the IDL command line. If you write the commands incorrectly in the file, you will get the same kind of errors you get if you had typed the commands incorrectly at the command line.

Suppose you had eight to ten image files that you wanted to view this way. You would have to open each image file, read the data into the *image* variable, and run this batch file to display the image in a window for each image. But suppose you wanted to automate this process still more.

You could, for example, automate the data reading and display process like this:

```
theseFiles = FindFile('*.img', Count=numFiles)
Print, 'Number of files found: ', numFiles
FOR j=0,numFiles-1 DO BEGIN
    OpenR, lun, theseFiles[j], /Get_Lun
    image = BytArr(512, 512)
    ReadU, lun, image
    Free_Lun, lun
    thisImage = BytScl(image, Top=199)
    LoadCT, 5, NColors=200
    s = Size(image, /Dimensions)
    Window, /Free, XSize=s[0], YSize=s[1]
    TV, thisImage
ENDFOR
```

But this file is not appropriate for a batch file because of the multi-line FOR loop. This kind of program syntax is difficult to write on an IDL command line without a lot of line continuation (\$) and command concatenation (&) characters. To automate the execution of IDL commands that contain multi-line control statements, it is better to use a main-level IDL program.

Writing a Main-Level IDL Program

A *main-level IDL program* is similar in many respects to a batch file, but it has several important distinctions. Like a batch file, a main-level program consists of a sequence of IDL commands. But unlike a batch file, these commands must be followed by an *END* statement. For example, here is the automated data reading and display program above written as an IDL main-level program module:

```
theseFiles = FindFile('*.img', Count=numFiles)
Print, 'Number of files found: ', numFiles
FOR j=0,numFiles-1 DO BEGIN
    OpenR, lun, theseFiles[j], /Get_Lun
    image = BytArr(512, 512)
    ReadU, lun, image
    Free_Lun, lun
```

```

thisImage = BytScl(image, Top=199)
LoadCT, 5, NColors=200
s = Size(image, /Dimensions)
Window, /Free, XSize=s[0], YSize=s[1]
TV, thisImage
ENDFOR
END

```

The most important distinction between a batch file and a main-level program is that the main-level program statements are compiled by the IDL compiler into a program unit *before* the code is executed. This is what makes it possible to have a multi-line control statement in an IDL main-level program.

If the code above is placed in a text file named *imageout.pro*, then the program module can be compiled and executed by typing this:

```
IDL> .RUN imageout
```

The main-level program is now resident in IDL's memory. There can only be one main-level program module resident in memory at any one time. To re-execute the main-level program module without re-compiling the code, use the *.GO* executive command, like this:

```
IDL> .GO
```

Executive commands (*.Go*, *.Compile*, *.Run*, etc.) can only be used at the main IDL level. They cannot be used in IDL procedures and functions. (Although there are ways to compile programs from within IDL procedures and functions. See "Special Compilation Commands" on page 237 for details.)

The main advantage of writing main-level IDL program modules is that the variables created in the program module are available at the main IDL level, which is where IDL commands are typed and interpreted. Another way to say this is that the IDL variables in a main-level program have *scope* at the IDL main level, where they can be used by other IDL commands.

More often you wish to isolate the scope of variables so that they don't take up an excessive amount of IDL's memory. Having a large number of main-level IDL variables is not a memory efficient way to program. For this reason, most IDL application programs are written with IDL procedures and functions.

Writing an IDL Procedure

An *IDL procedure* is similar to a main-level IDL program, with a few important distinctions. First of all, when you create an IDL procedure what you are really doing is creating another IDL command. You can think of this as building onto the IDL language. The commands you build can be used at the IDL command line and in IDL programs in much the same way built-in IDL commands are used.

Procedures look very much like IDL main-level programs, except that they begin with a *procedure definition statement*. The purpose of the procedure definition statement is to name the procedure (this is the command name, if you like) and to define any of the procedure's parameters. Procedure parameters can either be positional or keyword parameters. You will learn more about how parameters are defined in a moment.

Suppose, for example, that you wanted to make the main-level program above into an IDL command with the name *ImageOut*. You would write the procedure like this:

```

PRO ImageOut
theseFiles = FindFile('*.img', Count=numFiles)

```

```
Print, 'Number of files found: ', numFiles
FOR j=0,numFiles-1 DO BEGIN
    OpenR, lun, theseFiles(j), /Get_Lun
    image = BytArr(512, 512)
    ReadU, lun, image
    Free_Lun, lun
    thisImage = BytScl(image, Top=199)
    LoadCT, 5, NColors=200
    s = Size(image, /Dimensions)
    Window, /Free, XSize=s[0], YSize=s[1]
    TV, thisImage
ENDFOR
END
```

If this code was saved in a text file named *imageout.pro*, it could be automatically compiled and executed just by typing the procedure's name at the IDL command line. For example, like this:

```
IDL> ImageOut
```

Sometimes you want to explicitly compile a procedure or function before you run it. (For example, your code might have an error in it and you want to re-compile the code after you fixed the error and before you run it again.) To explicitly compile and run the code above, you would type:

```
IDL> .Compile imageout
IDL> ImageOut
```



Note that *imageout* in the compile statement above is the name of the file (IDL assumes a *.pro* file extension) you wish to compile. It is not enclosed in quotes, but it may be case sensitive, depending upon your operating system. The *Compile* command will compile all the program modules in the file, but it will not run any of the modules. (There is only a single program module in this example. You will learn more about how files are compiled in “Compiling and Running IDL Program Modules” on page 235.) The file does not have to have the same name as the procedure, but this is usually the case. The second statement above is the name of the IDL procedure or program module you wish to run.

Scope of Procedure and Function Variables

An important point about IDL procedures and functions is that the scope of IDL variables created inside the procedure or function is local. (*Scope* is the technical term for how widely accessible variables are to other commands.) That is to say, only commands inside the procedure and function can access variables created inside the procedure or function. For example, in the procedure above, it is impossible to know anything about the variables *theseFiles*, *thisImage*, *image* or *s* from the IDL command line, because these variables do not have scope at the main IDL level. Moreover, when IDL exits the procedure (by coming to the *END* statement in this case), the local variables are cleaned up and the memory they occupied is available to be used over again in IDL.

It would be extremely inconvenient if all variables had local scope, however, because then it would be impossible to communicate and pass information between procedures and functions or even between the IDL command line and procedures and functions. Thus, there are ways to extend the scope of a variable or to pass information into and out of procedures and functions. Usually information (variables, etc.) are passed by means of procedure and function parameters.

Creating a Positional Parameter

Positional parameters are defined on the procedure definition statement. There is no practical limit to the number of positional parameters you can define, but the order in which they are defined is crucial. The first positional parameter defined (“defined” means positioned to the right of the procedure name) is positional parameter 1, the next one defined is positional parameter 2, and so on.

For example, maybe you would like to be able to specify which directory the *ImageOut* procedure should use to look for the image files. You may wish to pass this information into the *ImageOut* procedure by means of a string parameter named *lookHere*. You can modify the *ImageOut* procedure above by adding the bold text statements below:

```
PRO ImageOut, lookHere
CD, lookHere
theseFiles = FindFile('*.img', Count=numFiles)
Print, 'Number of files found: ', numFiles
FOR j=0,numFiles-1 DO BEGIN
    OpenR, lun, theseFiles(j), /Get_Lun
    image = BytArr(512, 512)
    ReadU, lun, image
    Free_Lun, lun
    thisImage = BytScl(image, Top=199)
    LoadCT, 5, NColors=200
    s = Size(image, /Dimensions)
    Window, /Free, XSize=s[0], YSize=s[1]
    TV, thisImage
ENDFOR
END
```

To use this new procedure after you have made the modifications, you will have to recompile it. Then you might call it like this. (I am assuming a Windows machine in this case. This code is for illustration only.)

```
IDL> .Compile imageout
IDL> ImageOut, 'C:\RSI\myDataFileDir'
```

The string *C:\RSI\myDataFileDir* is copied into the parameter *lookHere* (this parameter is sometimes also called a *symbol* or *local variable*), where it is used by the *CD* command to change the working directory to the directory with this name.

The string can also be passed into *ImageOut* as a variable. For example, like this:

```
IDL> myImageDirectory = 'C:\RSI\myDataFileDir'
IDL> ImageOut, myImageDirectory
```

In this case, the variable *myImageDirectory* is said to be *passed by reference*. What this means is discussed in more detail below, but one thing it means is that the variable now has scope at the main IDL level and inside the procedure *ImageOut*. This means that if you change the contents of the variable inside the procedure, the change will be reflected in the contents of the variable back at the IDL command line. In other words, the variable *myImageDirectory* at the main IDL command line is now one and the same as the variable *lookHere* in the procedure *ImageOut*.

Another way to say this is that the variable has two names that refer to the same data item or physical address in IDL memory. (This is like a person being known as Joe in the United States but as José back home in Chile. Both names point to the same person, but that person is known in his local surroundings by only one name.)

But what if you forgot to pass a positional parameter in the call to *ImageOut*? What would happen if you just typed this:

```
IDL> ImageOut
```

In this case, nothing would be passed into the variable *lookHere*, so that variable would be undefined (which is a valid type for a variable) inside the procedure. Unfortunately, this will cause difficulties for you because the first statement in your procedure uses the variable *lookHere* as a parameter for the *CD* command. It is an error to use an undefined variable in the *CD* command in this way. Thus, it becomes imperative to know how your procedure was called.

Defining Optional or Required Positional Parameters

IDL supplies the *N_Params* command to tell you how many *positional* parameters (it does *not* count keyword parameters) your procedure was called with when it was used. The command is called like this:

```
numParams = N_Params()
```

Knowing how many positional parameters your procedure was called with gives you the opportunity to decide whether your parameters are going to be optional or required parameters for that particular procedure. For example, suppose you want to make the *lookHere* parameter a required parameter for this procedure. The first several lines of code in your procedure might look like this:

```
PRO ImageOut, lookHere
numParams = N_Params()
IF numParams EQ 0 THEN $
    Message, 'Must supply one parameter to this procedure.'
    CD, lookHere
```

In this case, the *Message* command generates an error condition, IDL stops executing commands in the procedure, and the message string is sent to the command log or output window. The result is similar to calling the *Plot* command with no data parameter.

But it probably makes more sense in this *ImageOut* procedure to make the *lookHere* parameter an optional parameter. If the parameter is not supplied, then the current working directory can be assigned to the variable *lookHere*. The code to implement this functionality might look like this:

```
PRO ImageOut, lookHere
numParams = N_Params()
IF numParams EQ 0 THEN CD, Current=lookHere
    CD, lookHere
```

In this case, if the *lookHere* parameter is not supplied in the call to *ImageOut*, then the *CD* command is used with the *Current* keyword to place the current working directory into the *lookHere* variable. (The *Current* keyword is an *output* keyword in this case. You will learn more about this in just a moment.)



A rule of thumb or convention used in defining parameters for IDL procedures and functions is that positional parameters are used for required parameters and keyword parameters are used for optional parameters. While this convention is often broken for positional parameters, it is almost never broken for keyword parameters. That is to say, you frequently find positional parameters that are optional parameters, but you rarely find keyword parameters that are required parameters.

Defining a Keyword Parameter

Like positional parameters, *keyword parameters* are also defined on the procedure definition statement. They can be mixed in among the various positional parameters if you like (they have absolutely no effect on the relative position of positional parameters), but they are usually defined after the positional parameters are defined.

If you look at the code for the *ImageOut* procedure, you will see that loading color table 5 is hard-coded into the program. But it might make more sense to allow the user to specify a color table and only use color table 5 as the default if the user doesn't have a preference. This is the perfect job for a keyword parameter. Keywords are defined with a syntax like this:

```
keywordName=keywordSymbol
```

The entity on the left-hand side of the equal sign is the keyword name. This is the name that is used when the keyword is used. The entity on the right-hand side is the variable that holds the value of the keyword inside the procedure or function. For example, here is how you can define a keyword named *ColorTable* for the *ImageOut* procedure:

```
PRO ImageOut, lookHere, ColorTable=thisColorTable
```

The keyword name *ColorTable* is what you will use when you call the *ImageOut* procedure with a color table value. For example, suppose you want to see these images using the Red-Temperature color table, or color table 3. You might call this procedure like this:

```
IDL> ImageOut, 'C:\Data', ColorTable=3
```

Using Keyword Abbreviations

The keyword name, *ColorTable*, does not have to be completely spelled out to be used. Keyword names only have to have enough letters to make them unique for the program module in which they are defined. Since this is the only keyword defined in the *ImageOut* module, the letter *C* is enough to define this keyword. The *ImageOut* program could be called like this:

```
IDL> ImageOut, 'C:\Data', C=3
```

Or, it could be called like this:

```
IDL> ImageOut, 'C:\Data', Color=3
```

Each of these three calls is equivalent.

The variable on the right-hand side of the keyword definition, *thisColorTable*, will hold the *value* of the keyword inside the program. In this case, it will hold the value 3.

You can now re-write the *ImageOut* code like this (changes are shown in bold type):

```
PRO ImageOut, lookHere, ColorTable=thisColorTable
numParams = N_Params()
IF numParams EQ 0 THEN CD, Current=lookHere
CD, lookHere
theseFiles = FindFile('*.img', Count=numFiles)
Print, 'Number of files found: ', numFiles
FOR j=0,numFiles-1 DO BEGIN
    OpenR, lun, theseFiles(j), /Get_Lun
    image = BytArr(512, 512)
    ReadU, lun, image
    Free_Lun, lun
    thisImage = BytScl(image, Top=199)
```

```

LoadCT, thisColorTable, NColors=200
s = Size(image, /Dimensions)
Window, /Free, XSize=s[0], YSize=s[1]
TV, thisImage
ENDFOR
END

```

But what if the variable *thisColorTable* is undefined? In other words, what if the user called the *ImageOut* procedure like this:

```
IDL> ImageOut
```

Since the *ColorTable* keyword was not used, there can be nothing in the variable *thisColorTable*. Another way of saying this is that the variable *thisColorTable* is undefined inside the procedure. If this is the case, then the *LoadCT* command two-thirds of the way down in the code above will fail, because it cannot accept an undefined variable as a parameter.

This situation should remind you of a situation you just encountered, in which you needed to know if a positional parameter was used in the call to *ImageOut*. Here you need to know if the keyword parameter was used in the call to *ImageOut*.

Defining Optional Keyword Parameters

Keyword parameters are (almost by definition) optional parameters. This means there is a good chance they will not be used in the call to the procedure or function for which they are defined. But the variable holding the value of the keyword will certainly be used in the code. This means that you will want to define default values for each and every keyword you define. In fact, if you don't, sooner or later your IDL program will fail.

But how will you know if you need to define a default value? You certainly don't need to define a default value if the user passed a value into the program. But how will you know if the user passed a value in? With positional parameters you used the *N_Params* command to give you this information. Unfortunately, *N_Params* works only with positional parameters. It provides no information at all about keyword parameters.

Many people think of this problem as "I want to know if the keyword was *used*, or not." As it happens, finding out if a keyword was "used" or not is more difficult and subtle than it might at first appear. We usually settle for something almost as good, but not the same. We ask, "Is the variable holding the value of the keyword *defined*, or not?"

Is the Keyword Defined?

For a keyword like *ColorTable*, which may be used to pass in many possible values, you use the IDL command *N_Elements* to tell you whether the variable *thisColorTable* is defined or not inside the procedure or function. *N_Elements* is used for other purposes in IDL, but in this instance you want to use a particular feature of the function: if the parameter to *N_Elements* is of type *undefined*, then the function returns a 0. It returns the true number of elements of any other type of parameter.

In this case, the first few lines of code in the *ImageOut* procedure, which will define a default color table value if one is not provided by the user, might look like this:

```

PRO ImageOut, lookHere, ColorTable=thisColorTable
numParams = N_Params()
IF numParams EQ 0 THEN CD, Current=lookHere
CD, lookHere
IF N_Elements(thisColorTable) EQ 0 THEN thisColorTable=5

```



Be sure you make a distinction between the *name* of the keyword (i.e., how it is used on the IDL command line) on the left-hand side of the keyword definition and the *variable* that holds the value of the keyword on the right-hand side of the definition. The parameter to *N_Elements* must be the keyword *variable*, not the keyword *name*.

This point is often misunderstood because some IDL programmers (including me) like to have the spelling of the keyword name and variable exactly the same. In other words, if I were writing this procedure for myself, I would write it like this:

```
PRO ImageOut, lookHere, ColorTable=colorTable
numParams = N_Params()
IF numParams EQ 0 THEN CD, Current=lookHere
CD, lookHere
IF N_Elements(colorTable) EQ 0 THEN colorTable=5
```

There is no correct method, but there is certainly a distinction. IDL will keep the distinction straight, so you are well advised to do the same. Remember that you are checking the keyword *variable*, not the keyword *name*. I like to have the keyword variable and name spelled the same because if I do I have less chance of making a typing mistake. I can *think* I am checking the keyword name if I like, when what I am really doing is checking the keyword variable.

Handling Keywords with Binary Properties

If you examine the code for *ImageOut* carefully, you will see that the image data is scaled before it is displayed. The particular line of code is this:

```
thisImage = BytScl(image, Top=199)
```

It is possible that the image data files may already be scaled, so that this is an unnecessary step. If this is the case, you may want to define a keyword, perhaps you could name it *Scale*, that when set allows the scaling to take place, but when not set allows the scaling to be skipped. Such a keyword has a binary property: it is either set or it isn't. Other keywords can also have binary properties. They are true or false, yes or no, 1 or 0.

IDL provides a special command to deal with keywords like this. It is the command *Keyword_Set*. Like *N_Elements*, the parameter to *Keyword_Set* will be the keyword variable, not the keyword name. But *Keyword_Set* acts a bit differently. If the parameter to *Keyword_Set* is of type undefined *or* if it has a value of 0, then *Keyword_Set* returns a 0. If the parameter is anything else at all, then *Keyword_Set* returns a 1. Thus, you can only get a 0 or a 1 returned from *Keyword_Set*.

Many IDL programmers misuse *Keyword_Set* and treat it as if it meant *keyword used*. It doesn't. Using it to determine if a keyword was *used* or not will eventually result in an IDL program that fails. Use it only with keywords with binary characteristics.

To make sure the scaling step is only invoked if the *Scale* keyword is set, the *ImageOut* code might now be written like this:

```
PRO ImageOut, lookHere, ColorTable=thisColorTable, $
      Scale=scaleIt
numParams = N_Params()
IF numParams EQ 0 THEN CD, Current=lookHere
CD, lookHere
theseFiles = FindFile('*.img', Count=numFiles)
Print, 'Number of files found: ', numFiles
FOR j=0,numFiles-1 DO BEGIN
  OpenR, lun, theseFiles(j), /Get_Lun
  image = BytArr(512, 512)
  ReadU, lun, image
  Free_Lun, lun
```

```

    IF Keyword_Set(scaleIt) THEN $
        thisImage = BytScl(image, Top=199) ELSE $
        thisImage = image
    LoadCT, thisColorTable, NColors=200
    s = Size(image, /Dimensions)
    Window, /Free, XSize=s[0], YSize=s[1]
    TV, thisImage
ENDFOR
END

```

Now if you wish the data to be scaled, you can call the *ImageOut* procedure like this:

```
IDL> ImageOut, /Scale
```

Remember that the syntax */Keyword* simply means *Keyword=1*. This shorthand way of specifying keywords is often used with keywords that have binary characteristics.

Passing Undefined Keywords by Keyword Inheritance

Look at your *ImageOut* program closely. You have a number of imbedded commands inside the program that take keywords. For example, you might want to specify a window title with the *Title* keyword to the *Window* command. Or you might want to change the order of the image display by setting the *Order* keyword on the *TV* command. Should you define a *Title* and *Order* keyword for this program, too?

I think you can see there would quickly be no end of it. If all possible keywords had to be defined, we would spend all our time defining keywords and no time getting our jobs done.

But there is another way, called *keyword inheritance*, to get these keywords into your programs. The way it works is that you define a keyword named *_Extra*. The underscore at the front of the name is essential. The first line of the *ImageOut* program will look like this:

```
PRO ImageOut, lookHere, ColorTable=thisColorTable, $
    Scale=scaleIt, _Extra=extra
```

When the program is called with keywords that are undefined for it, such as a *Title* or *Order* keyword:

```
IDL> ImageOut, /Scale, Title='Moon Images', /Order
```

IDL will create in the variable *extra* (you can, of course, give this variable any name you like) an anonymous structure, with each field in the structure set to the name of the undefined keyword name, and the value of each field the value of the undefined keywords. In other words, the *extra* structure will be created by IDL like this:

```
extra = {Title:'Moon Images', Order:1}
```

All you have to do is pass this *extra* structure along to the commands you wish to pass the undefined keywords to by using *their _Extra* keywords. For example, to pass the keywords along to the *Window* and *TV* commands, you would make these changes in your *ImageOut* program:

```
Window, /Free, XSize=s[0], YSize=s[1], _Extra=extra
TV, thisImage, _Extra=extra
```

Now, of course, the *TV* command doesn't know what to do with the *Title* keyword, and the *Window* command can't make heads or tails of the *Order* keyword. In versions of IDL before IDL 5.0, this would cause your program to crash. But in later versions of IDL, any inherited keyword that is not appropriate for a command is just quietly ignored. In other words, the *Window* command will respond to the *Title* keyword, but

not the *Order* keyword, and the *TV* command will do just the opposite, which is exactly what you had in mind.

Keyword inheritance is so useful that many IDL programmers just add an *_Extra* keyword to every procedure or function they write, even if they can't think of a reason for using it right away.

But let me give you a couple of words of caution. First, it is possible to have a keyword in the inherited structure that is the same as a keyword already used for the command. For example, suppose I was already assigning a title to the windows in the *ImageOut* program with the *Title* keyword. When I pass the *extra* structure along to the *Window* command it will have two *Title* keywords defined for it. This would cause an error if the command was executed, for example, on the IDL command line. But in the keyword inheritance mechanism, the first use of the keyword is ignored and the keyword value in the inherited structure is honored.

The second word of caution is more subtle. If you have keyword inheritance turned on by defining an *_Extra* keyword, you can sometimes be confused about why your program is not working the way you expect. For example, if you now call your *ImageOut* program like this, no errors will be reported, even though you have misspelled the *Scale* keyword:

```
IDL> ImageOut, /Scalee
```

At the same time, the images will not be scaled. The absence of error messages might convince you that the scaling portion of your code wasn't working, when what was really wrong is you misspelled a keyword. These kinds of errors can be hard to catch occasionally. Try to be aware of the possibility.

Creating Output Parameters

The keyword and positional parameters you have created so far in the *ImageOut* procedure have all been *input* parameters. In other words, information is coming into the procedure from some place outside of the procedure. But often you want to get information *out of* your procedure, too.

For example, the user of the *ImageOut* program may want to know the names of the files that were opened. Inside the procedure these file names are stored in the variable *theseFiles*. This variable is local to the procedure. Another way to say this is that the variable's scope is limited to the procedure itself. How can the user gain access to this variable from outside the procedure itself?

There are several ways. A common block, for example, can give wider scope to variables. But usually common blocks are not necessary. A simpler way is to use an *output parameter*.

Passing Information by Reference or by Value

Recall the discussion about positional parameters being passed by reference in “Creating a Positional Parameter” on page 211. It was said that parameters that are passed by reference have a scope larger than the local scope. That is to say that a parameter that is passed by reference in IDL is really—at a low level—a pointer to an address in IDL memory. This makes the parameter appear visible in the program unit that knows about it. The practical effect of passing data between program modules by reference is that operations done on the data by any program module that “knows” about the data affects the data in all modules in which it has scope.

The alternative way to pass information into and out of programs is to pass it *by value*. This means that a pointer to a memory address is not passed, but instead a copy of the

data is made and passed into the program module. Program modules that work on copies of the data cannot affect the original data at all.

In IDL, as it turns out, all variables are passed by reference. Anything else—and this includes such things as subscripted variables, system variables, expressions, structure de-references, and constants—are passed by value.

Here is a simple example to illustrate the point. Type the following small program into a text editor. The program resizes an image and displays it in a 150 by 150 display window. Save the program as *resizeit.pro*.

```
PRO ResizeIt, image
image = Congrid(image, 150, 150)
Window, 0, XSize=150, YSize=150
TVSCL, image
END
```

Now load the world elevation data set with *LoadData*, like this:

```
IDL> image = LoadData(7)
```

Use the *Help* command to examine the variable:

```
IDL> Help, image
IMAGE           BYTE      = Array(360, 360)
```

Now call the *ResizeIt* program with the image data as a parameter. Re-examine the image variable. It has changed to a 150-by-150 byte array.

```
IDL> ResizeIt, image
IDL> Help, image
IMAGE           BYTE      = Array(150, 150)
```

The *image* variable changed because it was passed into the *ResizeIt* program by reference. In other words, it had scope inside the *ResizeIt* program. In this case, the image variable served both as an input variable (information was passed into the program) and as an output variable (different information was passed out of the program). Variables can easily be input variables, output variables, or both, depending entirely upon how you write the code inside the IDL program.

Suppose you wanted to pass the *image* data into the program, but you don't like the fact that it gets changed inside the program. You can do one of two things: (1) rewrite the *ResizeIt* program so that it doesn't change the *image* variable, or (2) pass the *image* variable into the current program by value instead of by reference. In the first case, the program could be rewritten like this:

```
PRO ResizeIt, image
Window, 0, XSize=150, YSize=150
TVSCL, Congrid(image, 150, 150)
END
```

In the second case, you can use an expression for the positional parameter, like this:

```
IDL> ResizeIt, image + 0B
```

Sometimes you will have the opposite problem. You want something to change inside a procedure or function, but it doesn't. This is almost always because the parameter you passed into the procedure or function is not a variable. It may be *part* of a variable. For example, it might be a field of a structure. But structure de-references, system variables, and any kind of expression are all passed by value, not by reference. Only variables (and then, only *complete* variables) are passed by reference.

For example, suppose the image above were in a system variable that you had defined like this:

```
IDL> image = LoadData(7)
IDL> DefSysV, '!Image', image
```

If the *ResizeIt* program was called like this:

```
IDL> ResizeIt, !Image
```

there is no possibility of getting the newly sized image out of the procedure and into the system variable, since the system variable *!Image* is always passed by value and not by reference. The proper sequence of commands would be this:

```
IDL> thisImage = !Image
IDL> ResizeIt, thisImage
IDL> !Image = thisImage
```

To solve the problem of wanting to get the file names back from the *ImageOut* procedure, an output keyword might be appropriate. Keyword parameters, like positional parameters, can be passed by reference or by value. The new program code, with a *Filenames* keyword, might look like this, with the changes indicated by bold type:

```
PRO ImageOut, lookHere, ColorTable=thisColorTable, $
      Scale=scaleIt, _Extra=extra, Filenames=theseFiles
numParams = N_Params()
IF numParams EQ 0 THEN CD, Current=lookHere
CD, lookHere
theseFiles = FindFile('*.*img', Count=numFiles)
Print, 'Number of files found: ', numFiles
FOR j=0,numFiles-1 DO BEGIN
  OpenR, lun, theseFiles(j), /Get_Lun
  image = BytArr(512, 512)
  ReadU, lun, image
  Free_Lun, lun
  IF Keyword_Set(scaleIt) THEN $
    thisImage = BytScl(image, Top=199) ELSE $
    thisImage = image
  LoadCT, thisColorTable, NColors=200
  s = Size(image, /Dimensions)
  Window, /Free, XSize=s[0], YSize=s[1], _Extra=extra
  TV, thisImage, _Extra=extra
ENDFOR
END
```

You need do no more than this. If the user wishes to have the file names returned, he or she must simply use the keyword name and assign it to an IDL variable. For example, the code might look like this:

```
IDL> ImageOut, Filenames=myFiles
```

Even though the variable *myFiles* is probably undefined when the *ImageOut* procedure is called, it is defined as a string array when the *ImageOut* procedure returns. If the variable *myFiles* is defined at the time *ImageOut* is called, it would be re-defined by the *ImageOut* procedure and the original value would be lost.

Using Keyword Inheritance with Output Parameters



Note that the *_Extra* keyword inheritance mechanism described above passes keywords and their corresponding values into the program by *value*, and not by *reference*. Passing a variable by value means you pass a copy of the variable into the program, not the variable itself. Thus, changes that the program makes to the variable have local scope and not global scope. This makes it impossible to use *_Extra* for output keywords. As a result of this limitation, a new keyword inheritance mechanism that

passes keywords by reference was added in IDL 5.1. This mechanism uses a `_Ref_Extra` keyword definition mechanism. You can use either the `_Extra` or `_Ref_Extra` mechanism, but not both at the same time.

Is the Parameter Present?

Output parameters sometimes put you in a situation where you want to know if a positional or keyword parameter is *present*, or not. For example, maybe your program can perform an extremely time-consuming calculation on the data, but the results of the calculation are not always needed by the user of the program. You probably want to perform the calculation only if the user specifically requests the information by passing an output parameter.

To illustrate the problem more dramatically, how can you determine if the user of the `ImageOut` program called the program like this:

```
IDL> ImageOut
```

or like this:

```
IDL> ImageOut, Filenames=myFiles
```

The answer is, you can't tell. You could use `N_Elements` or `Keyword_Set` inside the `ImageOut` program to check the variable holding the value of the `Filenames` keyword (the variable `theseFiles`), but unfortunately the best you can do with these commands is tell if the variable is *defined* or not. That is not the same thing as whether the variable is *present* or not. If the variable `myFiles` is an undefined variable, then neither `N_Elements` or `Keyword_Set` can help you distinguish the two different ways of calling the `ImageOut` procedure shown above.

To address situations like this and to help people who want to know if a keyword was *used* or not, IDL introduced a new function in IDL 5 unfortunately named `Arg_Present`. (It is unfortunate name because the function doesn't really tell you whether the argument is present or not. Keep reading.)

The way `Arg_Present` works is that it will return a 1 if the parameter (variable) is passed as a keyword or positional parameter (in other words, the positional parameter or keyword was *used*) *and*—this is extremely important—the variable is *passed by reference*. It will return a 0 otherwise. If the keyword or positional parameter is used, but the parameter is passed by value, then `Arg_Present` returns a 0, just as if the parameter had not been used or is not present. Be very careful in how you use this new command.

Was the Parameter Used?

To tell unambiguously whether a parameter was *used* or not, you must combine the `N_Elements` function with the `Arg_Present` function inside your program, like this:

```
IF (N_Elements(myFiles) NE 0) OR Arg_Present(myFiles) THEN $  
    Print, 'FILENAME Keyword Was Used' $  
ELSE Print, 'FILENAME Keyword Was NOT Used'
```

Note that what you *can't* do is write a `Keyword_Used` function that you can call from within any other procedure or function. Can you think why this is impossible? Try writing such a function and see what difficulties you have.

If you can't figure it out, don't worry. I told you it was a subtle point. In over 12 years of IDL programming experience, I've never had a reason to require such a function. You can get along perfectly well with one or the other of the three functions `N_Elements`, `Keyword_Set`, or `Arg_Present`.

Writing an IDL Function

An IDL function is defined almost exactly like an IDL procedure is defined. In other words, positional and keyword parameters are defined in a function exactly as they are in a procedure. There are only two differences: (1) the function definition line starts with a *FUNCTION* declaration instead of *PRO* declaration, and (2) a function always returns a single specific IDL variable to the caller of the function. The IDL variable that is returned is called the *return value* of the function. The variable that is returned can have any valid IDL variable type or organization. It can be, for example, a large IDL structure variable. In practice, this means that all *Return* statements in functions must have one positional parameter, which is the return value of the function. (A function without a *Return* statement always returns an implicit scalar value of 0.)

For example, to make the *ImageOut* procedure a function instead of a procedure, you only have to make a single change, like that written in bold type below:

```
FUNCTION ImageOut, lookHere, ColorTable=thisColorTable, $  
    Scale=scaleIt, _Extra=extra, Filenames=theseFiles  
numParams = N_Params()  
IF numParams EQ 0 THEN CD, Current=lookHere  
CD, lookHere  
theseFiles = FindFile('.img', Count=numFiles)  
Print, 'Number of files found: ', numFiles  
FOR j=0,numFiles-1 DO BEGIN  
    OpenR, lun, theseFiles[j], /Get_Lun  
    image = BytArr(512, 512)  
    ReadU, lun, image  
    Free_Lun, lun  
    IF Keyword_Set(scaleIt) THEN $  
        thisImage = BytScl(image, Top=199) ELSE $  
        thisImage = image  
    LoadCT, thisColorTable, NColors=200  
    s = Size(image, /Dimensions)  
    Window, /Free, XSize=s[0], YSize=s[1], _Extra=extra  
    TV, thisImage, _Extra=extra  
ENDFOR  
END
```

Functions have a different calling syntax within IDL. Since functions always return a value, functions are called by placing the variable that will receive the return value on the left-hand side of an equal sign and the function call on the right-hand side. All positional parameters and keyword parameters to functions are included inside of parentheses, like this:

```
IDL> thisValue = ImageOut('C:\data', Filenames=myDataFiles)
```

Since *ImageOut* doesn't yet have a *Return* statement, the function returns an implicit 0 as the return value.

But suppose you wanted *ImageOut* to return the number of files that were opened and displayed. You might write the program like this, with changes indicated by bold type, below:

```
FUNCTION ImageOut, lookHere, ColorTable=thisColorTable, $  
    Scale=scaleIt, _Extra=extra, Filenames=theseFiles  
numParams = N_Params()  
IF numParams EQ 0 THEN CD, Current=lookHere  
CD, lookHere  
theseFiles = FindFile('.img', Count=numFiles)
```

```

Print, 'Number of files found: ', numFiles
FOR j=0,numFiles-1 DO BEGIN
    OpenR, lun, theseFiles[j], /Get_Lun
    image = BytArr(512, 512)
    ReadU, lun, image
    Free_Lun, lun
    IF Keyword_Set(scaleIt) THEN $
        thisImage = BytScl(image, Top=199) ELSE $
        thisImage = image
    LoadCT, thisColorTable, NColors=200
    s = Size(image, /Dimensions)
    Window, /Free, XSize=s[0], YSize=s[1], _Extra=extra
    TV, thisImage, _Extra=extra
ENDFOR
RETURN, numFiles
END

```

Then, if you wanted to display and print out all the names of the image files in a particular directory, you could write code similar to this:

```

IDL> numFiles = ImageOut('C:\data', Filenames=myDataFiles)
IDL> FOR j=0,numFiles-1 DO Print, myDataFiles[j]

```

It wouldn't matter if you had 10 files or 100 files, this code would still work perfectly.

More often functions are written to obtain the explicit result of some operation that is to be performed on a variable. For example, suppose you want to obtain the average value of a vector or an array. You might write an *Average* function in a text editor, like this:

```

FUNCTION Average, data
averageValue = Total(data)/N_Elements(data)
RETURN, averageValue
END

```

Use it like this:

```

IDL> thisData = [3, 5, 6, 2, 9, 5, 4]
IDL> avg = Average(thisData)
IDL> Print, avg
4.85714

```

The return value of a function can be used as a parameter to another IDL procedure or function, so you could write the three lines of code above as a single line, like this:

```

IDL> Print, Average([3, 5, 6, 2, 9, 5, 4])
4.85714

```

Square Bracket Notation and Function Calls

Note that the use of parentheses in IDL function calls can sometimes cause confusion in reading IDL code. It can be difficult to know just by examining the IDL code, whether a function is being called or an array is being subscripted with parentheses notation (which was discouraged in IDL 5.0, but still allowed). The code above, for example, exhibits this kind of confusion. In the absence of other information, *Average* could just as well be an IDL variable. (It is possible, but not always wise, to have a function and a variable both with the name *average*.)

To help make code more readable, Research Systems introduced in IDL 5 a “square-bracket” notation for subscripting variables. Thus, if you had defined a variable named *average* like this:

```
IDL> average = [4,6,3,8,2,1]
```

It could be subscripted like this:

```
IDL> number = average[3]
```

rather than like this, which was the only allowable way up until that time:

```
IDL> number = average(3)
```

The square bracket notation will distinguish the variable *average* from the function *Average*, which is required to have its parameters in parentheses. Using square-bracket notation for array subscripting will eliminate any confusion in your code between a subscripting operation and a function call.

Beginning with IDL 5.3, you can force square bracket notation by setting a compiler option. Normally this compiler option command is placed at the beginning of IDL procedures and functions. It causes the compiler to enforce square bracket subscript notation in that procedure or function.

```
Compile_Opt STRICTARR
```

Reserving Function Names with the Forward_Function Command

To be sure the IDL compiler recognizes confusing syntax as a function call and not as a subscripted array, the *Forward_Function* command can be used to pre-declare the function name and reserve it for the function. For example, if you wanted to reserve the name *Average* for the function above, you would type this:

```
IDL> Forward_Function average
```



Notice that the name of the reserved function is not in quotes and there is no comma after the *Forward_Function* call and before the name of the function.

Now, if you compile a program module and IDL comes across the word *average*, followed by parentheses, it will assume that this is a function call and not a subscripted variable reference.

Using Program Control Statements

Like other programming languages, IDL programs have the ability to use program control statements to control the order in which program statements are executed. Most of the time, a program control statement contains some kind of a Boolean test (this is usually an expression involving program variables), and some means of doing one thing if the test is *true* and some other thing if the test is *false*. Or, it contains a counter and some way to execute a statement over and over again, based on the value of the counter.

True and False Expressions in IDL

Whether an expression evaluates to *true* or *false* in IDL depends upon the type of data in the expression. A *true* condition in IDL is represented as:

- An odd, non-zero value for byte, integer, and long integer data types.
- Any non-zero value for floating-point, double-precision, and either single or double-precision complex data types.
- Any non-null string for string data types.

Any condition or expression that is not true in IDL evaluates as *false* using Boolean logic.



Note that pointers and objects are not evaluated as true or false. Rather, they are evaluated as *valid* or *not valid* by using the *Ptr_Valid* or *Obj_Valid* commands, respectively.

An example of a control statement of the first type is the classic IF...THEN...ELSE control statement. It is written like this in IDL:

```
IF test THEN statement1
```

where *test* is a Boolean expression, usually, and *statement1* is the kind of IDL command you might type at the IDL command line.



Note that *test* must always evaluate to a scalar value and it *must* be defined for the expression to be processed correctly by IDL. For example, this expression will fail because the variable *coyote* is undefined:

```
IDL> IF coyote EQ 'tricky' THEN Print, 'Missed him!'
```

An example of a control statement of the second type is the classic FOR loop. In a FOR loop you execute the same statement over and over again until the counter gets to a preset value. It is written like this:

```
FOR j=0,10 DO statement1
```

where *j* is the counter and *statement1* is the IDL command to be executed. For example, this *Print* command would be executed 11 times and print the values 0 to 10:

```
IDL> FOR j=0,10 DO Print, j
```

Making Multiple Statements Appear As Single Statements

Most control statements (see above) require single statements in their syntax. This is because all IDL commands must appear to be one command to the IDL interpreter. But often this is not what you want. For example, based on some test you may want to execute four different statements if the test is true and 15 statements if the test is false.

In IDL you can make it clear to the IDL interpreter that multiple statements should be treated as a single statement for the purpose of a control statement by using BEGIN and END statement blocks. The BEGIN tells the compiler that this is the start of a multiple line statement block, and the END concludes the block.

For example, the FOR loop above might be written like this if several statements were to be executed from within the loop:

```
FOR j=0,10 DO BEGIN
    statement1
    statement2
    statement3
    ...
END
```

While the END statement is all that is needed to end the BEGIN statement block, multiple END statements make the program hard to read and interpret. For that reason, IDL allows a number of different kinds of END statements, based upon the program control statement being used. For example, the code above would usually be written with an ENDFOR command in place of the END command, since it ends a statement block in a FOR loop:

```
FOR j=0,10 DO BEGIN
    statement1
    statement2
    statement3
    ...
ENDFOR
```

Valid END statements are ENDIF, ENDELSE, ENDFOR, ENDWHILE, ENDCASE, and ENDREPEAT. You will learn more about how to use these statements below.

The IF...THEN...ELSE Control Statement

One of the most widely used control statements is the IF...THEN...ELSE control statement. It works on the basis of a Boolean test or other expression that always evaluates to *true* or *false*. (See “True and False Expressions in IDL” on page 223 for more information.) Here is an example of such a statement:

```
IF (num GT 10) THEN index = 2 ELSE index = 4
```

The ELSE part of the statement is completely optional. The control statement can also be written like this:

```
IF (num GT 10) THEN index = 2
```

The syntax of an IF...THEN...ELSE control statement using multiple statements is just a bit tricky. Remember that this must appear to the IDL interpreter to be a single IDL command. If you were at the IDL command line trying to write a multiple line command for the IDL interpreter, you would have to use line continuation characters and line concatenation characters to accomplish your purpose. For example, a multi-line IF...THEN...ELSE control statement might be written like this on the IDL command line:

```
IDL> IF (num GT 10) THEN BEGIN $  
      index = 2 & $  
      num = 0 & $  
    ENDIF ELSE BEGIN $  
      index = 4 & $  
      num = -10 & $  
    ENDELSE
```

The line continuation and concatenation characters are completely unnecessary when this code is written in a file that is compiled before the code is executed. For example, this code would be written like this in a file that is part of a main-level program or IDL procedure or function:

```
IF (num GT 10) THEN BEGIN  
  index = 2  
  num = 0  
ENDIF ELSE BEGIN  
  index = 4  
  num = -10  
ENDELSE
```

The BEGIN and END statements imply the line concatenation and continuation for the compiler, as opposed to the IDL interpreter. *But*, the syntax of the command is not entirely arbitrary for the compiler, either.

For example, some programmers like to align their BEGIN and END statements so they can see at a glance what they refer to, like this:

```
IF (num GT 10) THEN  
  BEGIN  
    index = 2  
    num = 0  
  ENDIF ELSE  
  BEGIN  
    index = 4  
    num = -10  
  ENDELSE
```

But this code cannot be written this way in IDL. The reason is that the IF...THEN...ELSE control statement is broken up inappropriately and does not appear to the compiler as a single statement. If you wish to use the code format above, you will have to use line continuation characters in the code, like this:

```
IF (num GT 10) THEN $
BEGIN
    index = 2
    num = 0
ENDIF ELSE $
BEGIN
    index = 4
    num = -10
ENDELSE
```

The Conditional Expression

A new kind of expression, a *conditional* expression (?:) was introduced in IDL 5.1. It can often be used in place of an IF...THEN...ELSE control statement. For example, suppose that if the variable *num* is greater than or equal to 10, we want to set the variable *index* to 2. Otherwise, we want to set the variable *index* to 4. We could write such an expression using a conditional expression like this:

```
index = (num GE 10) ? 2 : 4
```

In this statement, the test (*num GE 10*) is evaluated first. If this test is *true*, the *index* variable is set equal to the first variable (this could itself be an expression) to the right of the question mark operator and to the left of the colon. If the test is *false*, the *index* variable is set equal to the variable (or expression) to the right of the colon.

The FOR Loop Control Statement

The FOR loop uses a counter to execute a statement or multiple statements a specified number of times. Here is an example of a FOR loop:

```
a = 1
FOR j=1, 10 DO BEGIN
    Print, j
    a = a * j * 2
ENDFOR
```

In this case, the counter *j* starts at 1 and goes to 10. Thus, the statements are executed 10 times. If you wanted *j* to increment by something other than 1, you can also specify the increment. For example, to have *j* increment by 2, you can type this:

```
a = 1
FOR j=1, 10, 2 DO BEGIN
    Print, j
    a = a * j * 2
ENDFOR
```

The WHILE Loop Control Statement

The WHILE loop executes a statement or statements while a test condition remains true. The test is evaluated at the start of the loop. An example of a WHILE loop looks like this:

```
WHILE (number LT 100) DO BEGIN
    index = amount * 10
```

```

        number = number + index
ENDWHILE

```

A WHILE loop evaluates the test at the start of the loop.

The REPEAT...UNTIL Loop Control Statement

The REPEAT...UNTIL loop is similar to a WHILE loop, except that the test condition is evaluated at the end of the loop rather than at the start of the loop. An example of a REPEAT...UNTIL loop is this:

```

REPEAT BEGIN
    index = amount * 10
    number = number + index
ENDREP UNTIL number GT 100

```

The BREAK Control Statement

New in IDL 5.4 is the BREAK control statement. It is used to break out of other control statements (FOR, WHILE, CASE, etc.) without completing them. For example, like this:

```

WHILE (number LT 100) DO BEGIN
    index = amount * 10
    number = number + index
    IF number GT 1000 THEN BREAK
ENDWHILE

```

The CONTINUE Control Statement

New in IDL 5.4 is the CONTINUE control statement. (Do not confuse this statement with the *.Continue* executive command.) The CONTINUE control statement is used to stop the current loop iteration, and move immediately to the next iteration in FOR, WHILE, and REPEAT UNTIL loops. For example, to loop from 1 to 100, while leaving out loop values from 10 to 20, type this:

```

a = 1
FOR j=1, 100 DO BEGIN
    IF j GE 10 AND j LE 20 THEN CONTINUE
    Print, j
    a = a * j * 2
ENDFOR

```

Note that the CONTINUE control statements are *not* allowed in IDL CASE and SWITCH control statements. This is in contrast, for example, to the C language, where such statements are allowed.

The CASE Control Statement

Sometimes you have a situation where a test condition can evaluate to a number of different solutions, not just to a Boolean solution. This is a perfect time to use a CASE statement.

One common use of a CASE statement is to evaluate events in a widget program and execute the appropriate code. For example, here is code that responds to button events in a widget program:

```

; What button caused the event?
Widget_Control, event.id, Get_Value=buttonValue

```

```

; Branch based on button value

CASE buttonValue OF
  'Sobel' : TVScl, Sobel(image)
  'Roberts' : TVScl, Roberts(image)
  'Boxcar' : TVScl, Smooth(image, 7)
  'Median' : TVScl, Median(image, 7)
  'Original Image' : TVScl, image
  'Quit' : Widget_Control, event.top, /Destroy
  ELSE: ; Do nothing at all.
ENDCASE

```

Notice that the ELSE case, which can be used to define a default action for those occasions when the test value doesn't correspond to any of the test conditions, is used in this code to let the event "fall through" the event handler. The ELSE case, if it is used, needs a colon after it, just like the other possible test conditions in the CASE statement. The ELSE case does not have to appear in a CASE statement, but if it is not there an error will be generated if the test value does not match any of the possible test conditions.

If you are going to use BEGIN and END statement blocks in a CASE statement, be careful. The ENDCASE statement can be used to end both the statement block and the end of the overall case statement. Sometimes multiple statement cases are written like this:

```

CASE thisTest OF
  0: x = 5
  1: BEGIN
    x = 5
    y = x * 2
  END
  ELSE: x = 0
ENDCASE

```



Note that a CASE statement always "breaks" when a case is met. In other words, once a CASE is evaluated as true, program execution goes to the next program statement after the CASE statement. This does not happen, for example, in a C program CASE statement.

The SWITCH Control Statement

The SWITCH statement, introduced in IDL 5.4, is used to select one statement for execution from several choices, depending upon the value of the expression following the word SWITCH. Each choice in a SWITCH statement block is preceded by an expression that is compared to the value of the SWITCH expression. SWITCH executes by comparing the SWITCH expression with each choice expression in the order written. If a match is found, program execution jumps to that choice and execution continues from that point, through the other choices in the SWITCH statement block.

```

SWITCH thisTest OF
  a * 3: x = 5
  b + 8: BEGIN
    x = 5
    y = x * 2
  END
  ELSE: x = 0
ENDSWITCH

```

Whereas CASE executes at most one statement within the CASE block, SWITCH executes the first matching choice and any following choice in the SWITCH block. Once a match is found in the SWITCH block, execution continues to any remaining choices. For this reason, the BREAK control statement is commonly used in SWITCH choices to force an immediate exit from the SWITCH statement block. The ELSE clause of the SWITCH statement is optional. If included, it matches any SWITCH expression, causing its code to be executed. For this reason, it is usually written as the last choice in the SWITCH statement.

The ELSE statement is executed only if none of the preceding choice expressions match. If an ELSE clause is not included and none of the choices match the SWITCH expression, program execution continues immediately below the SWITCH statement block without executing any of the SWITCH choices.

The GOTO Control Statement

Like all modern programming languages, IDL has a GOTO statement. But most good IDL programmers use it only when absolutely necessary. Liberal use of GOTOS will make even simple programs extremely difficult to follow.

The GOTO statement specifies a *program label*, which is a location in a file where program execution should resume when the GOTO is executed. For example, you might use a GOTO statement to break out of a FOR loop prematurely, like this:

```
FOR j=0,n DO BEGIN
    index = j * !PI * 5.165 * thisTestValue
    IF index GT 3000.0 THEN GOTO, jumpout
ENDFOR
jumpout: Print, index
```

The program label requires a colon. There may be an executable statement on the program label line, as in this case, although there doesn't have to be. If a statement is not available, IDL moves to the next executable statement following the program label in the file.

Error Handling Control Statements

There are several error handling control statements that are commonly used in IDL. Three are considered here. They are *On_IOError*, *On_Error*, and *Catch*.

The On_IOError Control Statement

The *On_IOError* control statement is similar to a GOTO statement in IDL. It allows you to specify a program label where program execution resumes if it is interrupted for any reason by an input or output error.

For example, the *On_IOError* statement might precede a section of code where a data file was read, like this:

```
On_IOError, Problem
OpenR, lun, filename, /Get_Lun
data = BytArr(256,256)
ReadU, lun, data
Free_Lun, lun
...
...
RETURN
Problem: Print, 'Problem reading data. Returning...'
IF N_Elements(lun) NE 0 THEN Free_Lun, lun
```

```
RETURN
END
```

Notice the colon after the program label, just like in the GOTO statement. There does not have to be a valid IDL statement on the program label line.

The On_Error Control Statement

The *On_Error* control statement indicates what IDL should do in the event of a programming error at program run time. Unlike the other control statements mentioned in this section, the *On_Error* control statement does not switch execution to a new IDL statement. Rather, it determines the action to take when the error occurs. Valid parameters to the *On_Error* command are 0, 1, 2, and 3. Table 15 shows the possible actions that can be taken when a programming error occurs.

Value	Action to Take
0	Stop immediately in the program context of the module that caused the error. This is the default action.
1	Stop immediately and return to the main IDL program level. The main IDL program level is the level where IDL commands are entered at the command line.
2	Stop immediately and return to the program module that called the module causing the error.
3	Stop immediately and return to the program module that registered the <i>On_Error</i> condition. (This may not be the current module.)

Table 15: *The possible actions that the On_Error statement can take when a programming error occurs.*

New users of IDL programs are often confused when an error occurs, because the default behavior (*On_Error* is set to 0) is to stop in the context of the program module that caused the error. Because they have an IDL prompt, many users think they are at the main IDL level. They are often dismayed when they type a *Help* command to see that all of their variables have “disappeared”.

What they are actually looking at are the local variables that reside inside the crashed program module. Typing the command *RetAll* (*Return All* the way back to the main IDL level) will restore their disappeared variables and get them back to where they think they are.



I often joke in my IDL programming courses that *RetAll* is the programmer’s most important tool. Seasoned IDL programmers almost always type *RetAll* as an automatic response to any program that fails to exit normally. It is a good habit to get into, especially if you are writing widget programs, which can behave very strangely indeed if you forget this important command.

To prevent confusion among the people who are using your program, it is sometimes a good idea to add an *On_Error, 1* to the top of your program code once your program has been debugged. Then, if and when the program crashes, IDL will execute its own implicit *RetAll* command and take the user back to the main IDL level.

The Catch Control Statement

The third type of error control statement, and probably the most powerful, is the *Catch* control statement. It works almost like the *GOTO* statement, but not exactly. The *Catch* control statement is called like this:

```
Catch, theError
```

where *theError* is the name of a program variable (it can, of course, have any name you like). When the statement is executed, IDL registers a *Catch* error handler for that particular program module. Only one *Catch* error handler can be registered for a module at any one time. At the same time that the error handler is registered, the program variable (*theError* in this case) is set to the value 0.

If a run time error occurs in the program module that has the *Catch* error handler registered for it, then the program variable is set to the proper error number (each programming error has its own associated number), and program execution is transferred to the *first* line of code *after* the *Catch* error handler.

In practice this line contains a block of error handling code. For example, suppose you were going to read a file from within your code. File reading is notorious for producing all kinds of program errors. You might want to protect that section of code with a *Catch* error handler, like this:

```

Catch, theError
IF theError NE 0 THEN BEGIN
    Catch, /Cancel
    Print, 'Problem reading data file. Returning...'
    IF N_Elements(lun) NE 0 THEN Free_Lun, lun
    RETURN
ENDIF

OpenR, lun, filename, /Get_Lun
data = BytArr(256,256)
ReadU, lun, data
Free_Lun, lun

Catch, /Cancel

```

Notice that the *Catch* error handler can be canceled at any time. In this particular example, it is canceled as the first line in the error handler code. This prevents an infinite loop if, heaven forbid, you introduce program errors in your error handling code! Later in the code a new *Catch* error handler could be established. There is no limit to how many *Catch* statements you can have in your code, but only one *Catch* error handler can be registered at any particular time.

Notice, too, that this *Catch* handler is being used to catch a file input/output programming error. What about *On_IOError*?

Error Handling Hierarchy

There is a hierarchy of error handling procedures and what happens inside a program module depends upon this hierarchy and which error handlers are registered for that particular program module. You can mix and match error handlers as you see fit. The hierarchy is as follows:

On_IOError > Catch > On_Error

If an *On_IOError* handler is registered for a particular program module, that takes precedence over any other error handler. If a *Catch* error handler is registered for a program module, that takes precedence over any *On_Error* condition.

In the absence of any error handling information, the default *On_Error* condition is in effect in a program module.

Note that errors can propagate backwards in the calling sequence and be caught by any program module containing a *Catch* error handler. For example, suppose module A, called from the main program level at the IDL command prompt, calls module B, which calls module C, which calls module D. If modules C and D have *On_Error* equal 2 (returned to the module that called you) error handling, and module B has a

Catch error handler, then an error in program module D will be handled by the *Catch* error handler in module B. Similarly, if module C had an *On_Error* equal 1 (return to the main program level) error handling, the error in module D would skip over modules B and A in going directly back to the main program level.

Reporting Errors

Errors, when they occur, are reported to the user via the *Error_State* system variable. (Note that *Error_State* replaces the *Error*, *Err_String*, *SysErr_String*, *SysError*, and *Msg_Prefix* system variables. You may still find these old system variables used in legacy code.) The *Error_State* system variable is a structure containing a number of important fields, but the two most useful in general programming are the *Code* and *Msg* fields. The *Code* field holds the error number associated with the error condition. (All errors in IDL are assigned a particular error number.) The *Msg* field contains the text of the error message.

For example, type these commands in IDL:

```
IDL> Print, xxx
IDL> Help, !Error_State, /Structure
```

You see the following information in your command output log:

```
% PRINT: Variable is undefined: XXX.

** Structure !ERROR_STATE, 7 tags, length=52:
  NAME      STRING      'IDL_M_UNDEFVAR'
  BLOCK      STRING      'IDL_MBLK_CORE'
  CODE      LONG        -167
  SYS_CODE  LONG        Array[2]
  MSG       STRING      'PRINT: Variable is undefined: XXX.'
  SYS_MSG   STRING      ''
  MSG_PREFIX STRING      '%'
```

You can see how the message prefix and message were used to construct the error message on the first line. Note also that this particular error is assigned a number or code of -167. You could use the *Code* field of the system variable to trap a particular type of error, if you know its code number.

Using *Error_State.Msg* will allow you to write a more general purpose *Catch* error handler than the one above. For example, you might report the error message to the user like this:

```
Catch, theError
IF theError NE 0 THEN BEGIN
  Catch, /Cancel
  Print, !Error_State.Msg
  RETURN
ENDIF
```

Rather than just printing the error out into the user's command log window, you sometimes want to make more of an issue of it and stop all program execution until the user acknowledges that an error has occurred. The *Dialog_Message* command is useful for this purpose, since it is a modal widget program that must be dismissed by the user before program execution can continue. For example, the *Catch* error handler can be written like this:

```
Catch, theError
IF theError NE 0 THEN BEGIN
  Catch, /Cancel
  ok = Dialog_Message(!Error_State.Msg, /Error)
```

```
RETURN
ENDIF
```

Keywords on the *Dialog_Message* command can change the way the platform-dependent dialog appears to the user. No keywords at all presents what is called a “warning” dialog. The *Error* keyword used above produces an error dialog. The *Information* keyword will produce an informational dialog, as shown in Figure 92. And, of course, it is possible to put any message text you like as the argument to the *Dialog_Message* command. You are not restricted to the error message text at all. In the examples shown below the error message was the string “Error Message Here”.



Figure 92: The different types of dialogs available with *Dialog_Message*. These dialogs are from a Windows computer. Yours may look different from the ones illustrated here.



Note that prior to IDL 5.3 the *Dialog_Message* command could only be used if the current graphics device supported widgets. This limited its usefulness in IDL programs that were designed to run in a graphics device independent way. Error handlers had to be written to check whether widgets were supported (the system variable *!D.Flags* was used for this purpose) and then call either *Dialog_Message* or the *Print* command. (The *Message* command, described below, was also sometimes used.) To circumvent these problems and to incorporate other features, the *Error_Message* program was written. This program is among those you downloaded to use with this book. It is described in more detail below.

Generating Errors

There are two ways to generate an error condition in IDL: (1) make an error (bad), or (2) generate your own error condition as a result of some test (good). Well-written programs try to avoid the first by making extensive use of the second. Your users will appreciate this because you will have *very* helpful error messages. Won’t you? I mean, the alternative is to just let errors occur and allow IDL to supply the (sometimes cryptic or unhelpful) error messages in *!Error_State.Msg*.

You can generate your own error condition with the *Message* command in IDL. The text argument will be your especially helpful error message:

```
Message, 'Whoops, an error occurred. Sorry. :-)'
```

The text passed to the *Message* command will be placed into the *!Error_State.Msg* field, and the *Code* field will be set to -7. (The *Message* command has no provision for setting the error code number, unfortunately). The *Message* command is often used in conjunction with a *Catch* error handler: For example, suppose this is your *Catch* error handler code in an IDL program module named *Display_Image*:

```
Catch, theError
IF theError NE 0 THEN BEGIN
  Catch, /Cancel
  ok = Dialog_Message(!Error_State.Msg, /Error)
  RETURN
ENDIF
```

And suppose you were checking to see if an *image* variable was really a 2D array, like this:

```
ndims = Size(image, /N_Dimensions)
IF ndims NE 2 THEN BEGIN
    Message, 'Image argument must be 2D array.'
ENDIF
```

If the user passed a 24-bit image into the program as an argument (a 24-bit image is a 3D array), the error message would appear to the user as in Figure 93, below.



Figure 93: The *Dialog_Message* command displaying the error message generated by the *Message* command.

Note that the name of the program module in which the error occurred is placed before the error message. This is why you want to use the *Message* command to print the error message to the display, rather than, for example, the *Print* command. In other words, this capability is part of the error handling machinery built into IDL and can be accessed by the *Message* command.

Tracing Errors

Sometimes it is important to know where an error has occurred in your program. Especially if you want to fix it. You can use the *Traceback* keyword to the *Help* command after an error has occurred, for example, to find this information.

```
IDL> Help, /Traceback
```

And you can find out which program module generated the error by examining the calling stack with the *Help* command's *Calls* keyword:

```
IDL> Help, Calls=callStack
```

This works well for the case in which program operation stops when an error occurs (e.g., when *On_Error* is set to 0). But it often is inadequate when you are catching errors that may have occurred in a module other than the one which registered the *Catch* command.

For this reason (and others) I have written the *Error_Message* command, which is among the programs you downloaded to use with this book. Called without positional parameters, the *Error_Message* command uses either the *Dialog_Message* or the *Message* command with the *Informational* keyword set, depending upon whether widgets are supported by the current graphics device, to print the *!Error_State.Msg* field. Or, you can supply your own text message as an argument to the command. For example, the error handler above can be re-written more simply as this:

```
Catch, theError
IF theError NE 0 THEN BEGIN
    Catch, /Cancel
    ok = Error_Message(/Error)
    RETURN
ENDIF
```

Another feature of *Error_Message* is that it proceeds the “name” of the module it is called from to the error message. This can be immensely helpful in complicated widget or object programs with many program modules. It permits writing the same error handling code in each module, while still knowing where the error is coming from.

One consequence of *Error_Message* adding the name of the program module it is called from to the message is that if you are going to use the *Message* command to generate errors that will be caught and handled by *Error_Message*, you must be sure to set the *NoName* keyword. Failure to do so will cause the module name to appear twice in the error message.

```
Message, 'Whoops. You caused an error.', /NoName
```

The *Error_Message* command is also useful for generating a traceback of where the error occurred. Simply set the *Traceback* keyword to turn this functionality on:

```
ok = Error_Message(/Traceback)
```

A nicely formatted error traceback will be written to the command log window.

Compiling and Running IDL Program Modules

There are three IDL executive commands that can be used to compile program modules. These are *.Run*, *RNew*, and *Compile*. Historically, the oldest of these executive commands is the *.Run* command. In the early days of IDL there was no such thing as a procedure or function. IDL scripts were simply collections of IDL commands. The *.Run* command was used to compile and run these scripts, which are today called IDL main-level programs.

Since main-level programs produce all their variables at the main IDL level, there was always some concern about memory allocation, especially as one main level program after the other was compiled, each creating its own main-level variables. Thus, the *RNew* command was eventually introduced. The *RNew* command is similar to the *.Run* command, except that it causes all existing main-level variables to be deleted before the main-level program is compiled and run.

Over time, procedures and functions were introduced to IDL. As a practical matter, the *.Run* and *RNew* commands were used to compile these new program modules, too. But these commands didn’t run these compiled modules, they just compiled the modules. It has always confused new IDL users that the *.Run* command mostly didn’t *run* anything (except the occasional main-level program). Thus, the *Compile* command was introduced in IDL 4 to do what had been done with the *.Run* command for so many years. That is, to compile IDL procedures and functions.

Today, it is accepted practice to use the *Compile* command to compile procedures and functions and the *.Run* (or *RNew*) command to compile and run main-level programs. But the commands you can use to compile an IDL procedure named *CIndex* in the file *cindex.pro* could easily look like any of these:

```
IDL> .Run cindex
IDL> .RNew cindex
IDL> .Compile cindex
```

Notice that the file name is not enclosed in quotes and that the *.pro* file extension is not required with any of these three commands. The *.pro* file extension is assumed, however. (If you use a different file extension for your files, you must specify the file name in the commands above *with* the file extension.) Almost all IDL program files end with a *.pro* file extension. On machines that are case sensitive (e.g., UNIX machines), then the file name is also case sensitive. In general, on case sensitive

machines it is a good idea to keep all file names in lowercase letters. This is essential for the automatic compilation of files. (See “Rules for Compiling IDL Program Modules Automatically” on page 236 for more information on automatic compilation.)



Note that to run a program module that has been compiled, you must use the module’s name in an IDL command. For example, to use the *CIndex* procedure that was compiled above, you would type:

```
IDL> CIndex
```

The module’s name is not case sensitive inside of IDL.

Rules for Compiling IDL Program Modules Automatically

There are several rules for compiling program modules automatically that have implications for the ordering of program modules in a file.

Program modules in a file will be compiled one after the other until certain conditions are met. The rules below specify what those conditions are.

Rule 1: The file will be compiled until a main-level program is encountered, in which case compilation stops as soon as the main-level program is compiled, and the main-level program is run.

This rule implies that there can only be one main-level program module in a file, and—if you intend for all the program modules to be compiled—that the main-level program module should be the *last* program module in the file.

Rule 2: The file will be compiled until a program module is encountered which has the same name as the file, in which case compilation stops after that module is compiled and the program module is run.

This rule implies that if you want all program modules to be compiled that the file should have the same name as the *last* program module in the file. In other words, if your last program module is a function named *ImageOut*, then the file should be named *imageout.pro*.

Rule 3: The file will be compiled until the end of the file is reached or until one of the other rules is engaged.

This rule implies that if the file does not have either a main-level program or a program module with the same name as the file, then all the modules in the file will get compiled, but none of the modules will be run or executed.



Note that if you use the *.Run*, *.RNew*, or *.Compile* executive commands to compile your program file that every program module in the file will be compiled, including main-level programs.

Structuring Program Files

IDL will compile and run a procedure or function automatically when the name of that procedure or function is included in an IDL command issued either at the IDL command line or in a line of IDL code, provided that:

- The file containing the source code of the procedure or function is in the current working directory or one of the directories specified by the *!Path* system variable, and that,
- The name of the procedure or function is the same as the file name (without the *.pro* file extension). On case sensitive operating systems (e.g., UNIX) the file name must also be spelled in all lowercase characters.

What this means in practice is that program modules important enough to be called from the IDL command line or used in more than one other procedure or function should be placed in their own files and the files should be given the names of the program modules (with a *.pro* file extension). Any other modules that are included in the file should be located before the main program module and should be utility routines for the main program module.



Note that IDL system routines (e.g., *Plot*, *Surface*, etc. always have precedence over any other command name. You should not try to give your programs the same name as built-in IDL commands.

Special Compilation Commands

There are two special commands that can be used in IDL program modules to help compile other program modules. These are the *Resolve_Routine* and *Resolve_All*.

The *Resolve_Routine* command takes the name of an IDL program module as a parameter and compiles the file of the same name. In other words, it acts as if the *.Compile* command had been used to compile the file. The advantage of the *Resolve_Routine* command is that it can be used inside an IDL program module, whereas the *.Compile* command can only be used at the IDL command line. For example, to compile all the program modules in the *cindex.pro* file, you would type this:

```
IDL> Resolve_Routine, 'cindex'
```

If the program module is a function rather than a procedure, then the *Is_Function* keyword should be used, like this:

```
IDL> Resolve_Routine, 'congrid', /Is_Function
```

The named routine is compiled whether or not it had been compiled previously.

The *Resolve_All* command is similar, except that instead of compiling just a specific file, it iteratively searches through IDL memory for any uncompiled program module references and compiles those files as well. *Resolve_All* uses the normal rules for automatic compilation. This means that *Resolve_All* cannot find and compile uncompiled modules if they cannot be compiled automatically. Note, too, that *Resolve_All* cannot find structure and object definition files automatically. These files have an “*_define*” as part of their names. (Note the two underscore characters.) These files will have to be compiled by hand if you want to include them in a *save* file, as described in the next paragraph.

The *Resolve_All* command is convenient when you are preparing a *save* file for an application that you wish to run on another machine that may not have the same set of program files or directory structure as your machine. For example, if your application program is called *BigApp*, you might want to use *Resolve_All* to compile and save all the library and program files associated with the application, like this:

```
IDL> .Compile bigapp
IDL> Resolve_All
IDL> Save, /Routines, File='bigapp.sav'
```

Then, all the user of your program has to do is restore the *save* file and all of the necessary program modules are compiled and ready to be used. The source code files do not have to exist on the user’s machine.

```
IDL> Restore, 'bigapp'
IDL> BigApp
```



Note that compiled procedures and functions saved with the *Save* command are *not* guaranteed to be compatible in different versions of IDL. (Data or variables in save

files are always guaranteed to be compatible from one IDL version to another.) End users will, in general, have to restore the files using the same version of IDL that you used to save them. This can also be important if you upgrade to a new version of IDL.

Chapter 9

◆ Discovering the Possibilities ◆◆◆



Writing an IDL Graphics Display Program

Chapter Overview

The purpose of this chapter is more ambitious than most. Even though IDL is a programming language, it is impossible to find anywhere in the official IDL documentation *how* to write an IDL program. I don't mean to suggest there is only one right way. But anyone who has looked over the shoulders of as many IDL programmers as I have knows there is definitely a distinction between a good IDL program and one that is not so good. As someone who spends a lot of time with people who are trying to learn IDL for the first time, I see a lot of not-so-good programs.

I am convinced the problem is lack of information. Most people using IDL are, after all, scientists, not computer programmers. They are bright and they are trying to get their work done. They are not trying to write elegant computer programs.

But, still... If only a couple of simple principles were followed, their programs would be so much better and so much more useful to them. This chapter is an attempt to specify what those principles might be, while at the same time showing you how to assemble and use many of the techniques that have been discussed in this book.

The task I have set for myself is to show you how to write a reasonably complex graphics display program that you can use from the IDL command line. But I want to write this program in such a way that the output can be displayed in a resizable graphics window, printed directly from the IDL command line, or made into a PostScript file with little or no effort. The program should use colors in an intelligent way that doesn't depend upon the visual depth or color decomposition state of the output device. And finally, it should be simple to add a graphical user interface to this program, so that it can be used by someone unfamiliar with it.

Most programmers know that the best way to learn programming is by understanding code that has been written by others. But most of the time this is a daunting task, given the general lack of documentation in code and the absence of a mentor who can explain the unfamiliar elements of the code to you. My purpose here is not just to show you how to write a program, but to explain why the program is written in this particular way. Writing programs always involves making choices. The choices we make play a pivotal role in how useful the program is to us, and how easy it is to maintain and extend the program over time. In other words, we can make both helpful and non-helpful choices. By the end of the chapter you should be well on your way to understanding the distinction.

The HistoImage Program

The program I want to write will be named *HistoImage*. It is quite simple. It will display an image with axes around it. Above the image will be a colorbar that will indicate the image values. And above the colorbar will appear a histogram plot of the image pixel values. In other words, the histogram will show how many pixels in the image correspond to a particular image value. You see an illustration of how the program will appear in Figure 94. The complete listing of the program source code can be found “HistoImage Program” on page 409. And the *histoimage.pro* file is among the program files you downloaded to use with this book.

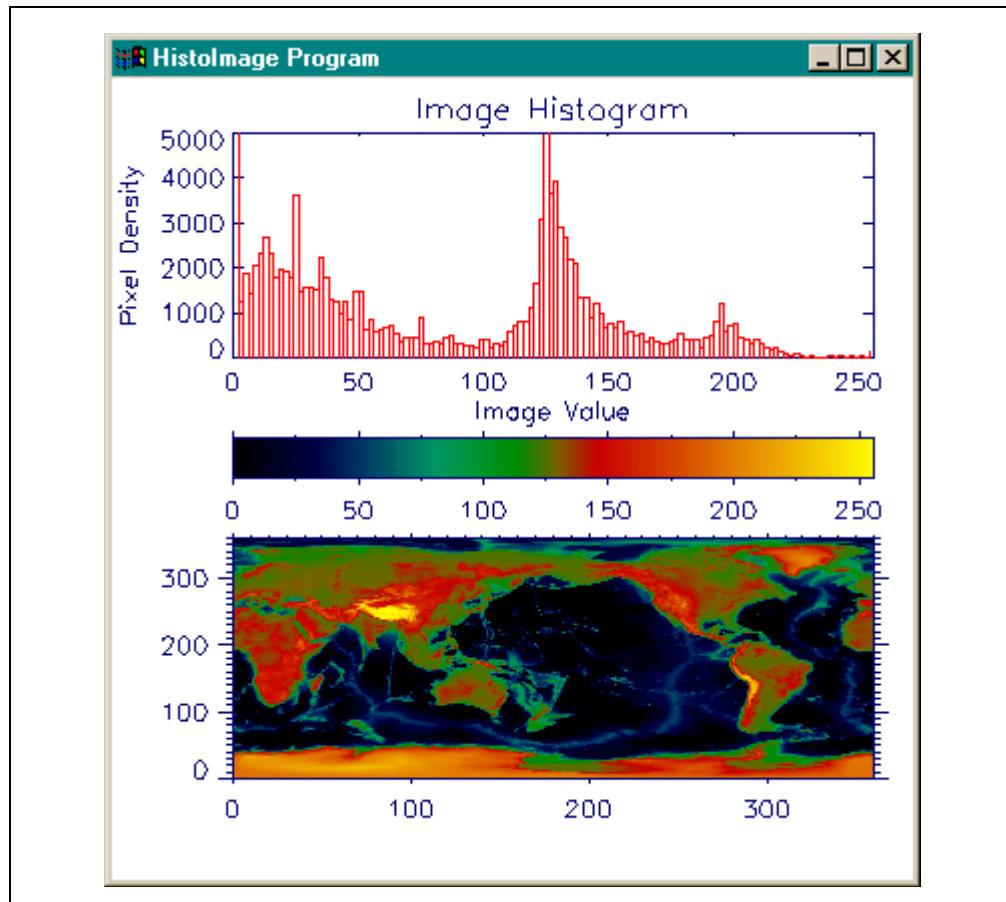


Figure 94: The *HistoImage* program consists of a histogram plot, a colorbar, and an image surrounded by axes.

Writing the Procedure Definition Statement

The *HistoImage* program will be a procedure, so I start by defining the procedure definition statement. This is where all the positional and keyword parameters are declared. A rule of thumb for parameters is that any parameter required for the program to run will be a positional parameter, and anything else will be a keyword parameter. This means, of course, that I will have to define default values for all of my input keyword parameters.

It also means that I seldom have in mind all of the keyword parameters I am going to need when I start coding the program. Most of the time I start with a few obvious ones and add others as it occurs to me I need them. I do try to give the users of my programs as much flexibility as possible. In particular, I like to give them as much

control as I can over how things are going to look, since I know users almost always have a different aesthetic sense than I do. So I will almost always have keywords that will allow the user to choose colors.

Here is what the procedure definition statement will look like:

```
PRO HistoImage, $
    image, $
    AxisColorName=axisColorName, $
    BackColorName=backcolorName, $
    Binsize=binsize, $
    ColorTable=colortable, $
    DataColorName=datacolorName, $
    Debug=debug, $
    _Extra=extra, $
    ImageColors=imagecolors, $
    Max_Value=max_value, $
    NoLoadCT=noloadct, $
    XScale=xscale, $
    YScale=yscale
```

The first and only positional parameter, *image*, is the image data that will be passed into the program. I am going to have to check whether this is a 2D array, since it doesn't make much sense to calculate the histogram of a 24-bit or 3D image, but I will delay this for a moment.

You see three “color” name keywords in the list: *AxisColorName*, *BackColorName*, and *DataColorName*. These will be the names of colors to use for the axes and other annotations, the background, and the data, respectively. I am going to use color names for these because one of my goals for the program is to use colors that are independent of the color decomposition state or depth of the display. I know that the *GetColor* program described in “Obtaining Device Independent Colors” on page 87 has the ability to give me a such a color if I ask for one of the 16 colors it knows about by name. By allowing the user to select these colors themselves, I give them some control over how things look on the display. I also allow the user to specify a color table index number with the *Colortable* keyword. The image data will be scaled into indices loaded by the color table.

Image colors almost always present a dilemma for me. On the one hand, I really want to set the image colors up correctly, especially if I am calling this program from the IDL command line. But on the other hand, I often prefer that color manipulation be done *outside* the program code. For example, in widget programs image colors are often controlled by a color table changing tool like *XLoadCT* or *XColors*. Loading a color table inside a graphics display program will often interfere with the colors that are being manipulated elsewhere.

I compromise in this program by defining two additional keyword parameters: *NoLoadCT* and *ImageColors*. *NoLoadCT* will be a flag that if set will prevent the program from loading the color table specified by the *ColorTable* keyword. In other words, I will have a way to turn color table loading off from outside the program, if this is what I choose to do. *ImageColors* will be an *output* keyword that will allow me to learn from outside the program how many colors the image data should be scaled into. This is essential information if I am to manage the colors from outside the program. (Here I will be loading image colors starting at color table index 0. If this were not the case, I might also define a keyword, say *BottomIndex*, that would indicate the bottom of the image color indices.)

The *BinSize* keyword is a *Histogram* command keyword. I will use the *Histogram* command inside this program to calculate the histogram plot and I may want the user

to be able to configure the properties of this command. Note that I don't provide all of the keywords that are available for *Histogram*, only those that I specifically want the user to manipulate. Similarly, the *Max_Value* keyword is a *Plot* command keyword I often find useful in histogram plots, so I define it here.

It is entirely possible (especially with the *Plot* command) that I will want to manipulate other *Histogram* or *Plot* or *TVImage* keywords that are not defined here. For example, I may want different axes annotations or tick marks. For this reason, I include an *_Extra* keyword to take advantage of keyword inheritance, which was described in "Passing Undefined Keywords by Keyword Inheritance" on page 216.

There are two keywords, *XScale* and *YScale*, that will be used to define the range or scale of the axes that surround the image. These input keywords will be two-element arrays defining the minimum and maximum extent of the axes.

Finally, there is a *Debug* keyword, which I intend to use in the error handler portion of the code to force a traceback of where the error occurred. I typically don't like to write error tracebacks into the command log window unless the user explicitly asks for such a thing. Using a *Debug* keyword gives me an easy way of debugging my code without frightening unfamiliar users with long lines of error text.

Writing the Error Handling Code

The next step in writing a program is to add some error handling code. Since I haven't given a great deal of thought to the kinds of errors that might occur in the program, and since I know that users will discover errors that I didn't anticipate anyway, I'll write a general purpose *Catch* error handler. (The *Catch* error handler is described in "The Catch Control Statement" on page 230.) I'll use the *Error_Message* program, which is among the programs you downloaded to use with this book and is described on page 234, because I want to take advantage of its device-independent nature.

Error_Message uses *Dialog_Message* to report the error if it is running on an output device that supports widgets and *Message* if the device does not support widgets.

Error_Message can also provide good error traceback information if it is required. The code looks like this:

```
Catch, theError
IF theError NE 0 THEN BEGIN
    Catch, /Cancel
    ok = Error_Message(!Error_State.Msg + ' Returning...', $
        Traceback=Keyword_Set(debug))
    RETURN
ENDIF
```

Notice that I cancel the *Catch* error handler as the first step in the error handling part of the code. I do this because I make too many typing errors during development and I sometimes have errors in my error handling code. This will cause IDL to go into an infinite loop. Call this good defensive programming if you can't think of a gentler explanation.

Note that I am adding a bit more information (the string *Returning....*) to the normal error message. I just want the user to know what I am doing with the error. And notice that the *Traceback* keyword is set for *Error_Message* only if the user set the *Debug* keyword when he or she called the *HistoImage* program. Since the *Debug* keyword can either be on or off, I set its value with *Keyword_Set*, which only returns a 0 or 1.

Checking for Positional and Keyword Parameters

Immediately after the error handling code, goes the code that checks for all the required and optional parameters. Checking is essential (at least for input parameters)

because you will be using the variables somewhere in the code to follow and it is not possible to use undefined variables in IDL expressions. (Methods for checking positional and keyword parameters are discussed in “Writing an IDL Procedure” on page 209.)

Checking for the Image Positional Parameter

I have mentioned earlier that my rule of thumb is that any required parameter is a positional parameter, and any optional parameter is a keyword parameter. I can’t do much in a program that calculates the histogram of an image and displays it above the image without an image! Hence, the *image* parameter should be a required positional argument.

But I’ve never been a person who cared much for arbitrary rules, and I’m going to choose to break this one right away by being a little kinder to the user and allowing this positional parameter to be an optional parameter. Why? Because I don’t think users should be penalized too harshly if they don’t know how to use a program. It is my job to explain it to them. So I will select and use an image for them. If they want another image, they can read the program documentation (you did write this, didn’t you?) to see how to use their own image.

The code will look like this:

```
IF N_Elements(image) EQ 0 THEN image = LoadData(7)
```

Note that I used *N_Elements* to check the *image* parameter, rather than *N_Params*, which you might have expected me to use for a positional parameter. Recall that *N_Elements* tells me if the *image* parameter is defined or not, whereas *N_Params* tells me the number of positional parameters the procedure was called with. (See “Defining Optional or Required Positional Parameters” on page 212 for more information.) Some inexperienced users are sure to call the *HistoImage* program with a single positional parameter that is an undefined variable. By using *N_Elements* I account for this eventuality.

If an *image* parameter is not supplied, I simply load the world elevation data set with the *LoadData* program you downloaded to use with this book.

Now that I have an *image* parameter, I will check to be sure it is a 2D array, since that is also a requirement for the histogram data to make sense. I can use the *Size* command with the *N_Dimensions* keyword set to determine the number of dimensions of the image. (Note that the *N_Dimensions* keyword to the *Size* command is a fairly recent introduction to the programming language. If you are using a version of IDL prior to IDL 5.2, you can obtain the same information from the *Size* command directly. See your on-line help for details.)

```
ndim = Size(image, /N_Dimensions)
IF ndim NE 2 THEN $
    Message, '2D Image Variable Required.', /NoName
```

The *Message* command will “throw” an error, which will be handled by my *Catch* error handling code. Note that the *NoName* keyword is set to prevent *Error_Message* from reporting the name of the program twice.

Checking for Keyword Parameters

I’m now ready to check for optional keyword parameters. I check the two keywords, *BinSize* and *Max_Value* I’m planning to use with the image histogram first. I expect most of the images used with this program will be byte data, but I can’t rely on this to be the case. Nor do I want to restrict the user to byte image data. But I do want the histogram plot of byte and, say, float image data to look the same. Thus, I am going to have 128 bins unless the user tells me something different. This will give me a

reasonably good looking plot almost always. I will have to calculate the bin size appropriately for this. The code will look like this:

```
IF N_Elements(binsize) EQ 0 THEN BEGIN
    range = Max(image) - Min(image)
    binsize = 2.0 > (range / 128.0)
ENDIF
IF N_Elements(max_value) EQ 0 THEN max_value = 5000.0
```

The bin size will either be 2, or it will be the image data range divided by 128, whichever number is larger. I am making the assumption here that I am not going to have floating point image data, that ranges from 0.0 to 1.0, for example. My program will look lousy with such data, but the chances seem so low of this happening that I am willing to chance it. And, anyway, if the user did have image data like that, they could always specify an appropriate bin size for viewing the histogram.

Note the use of the IDL “greater than” operator ($>$). This operator returns the larger of the two values being compared. Note, too, the parentheses about the value I want to compare to the right of the operator. Without the parentheses 2.0 would be compared to the range, and then *that* value would be divided by 128. Not what I want at all! This happens because the *greater than* operator has the same order of precedence as the *division* operator. This is a common kind of error to make in IDL programs.

Another common error occurs within the parentheses. Notice I have made the number 128.0 a floating point number by adding a decimal point to the number. You might easily make the mistake of writing this number as an integer (e.g., 128). Then, for byte or integer image data, you would be dividing an integer value (the *range*) by another integer value. This might easily give you a consistent bin size of 0. A serious error.

The *Max_Value* keyword variable is assigned a value of 5000, a value I know works with most of the example image data sets distributed with IDL.

Next, I can test for the *XScale* and *YScale* keyword values. If these are not supplied, I’ll use the dimensions of the image for scale values. I’ll also test to be sure the values are two element arrays. If not, I’ll issue error messages. The code will look like this:

```
s = Size(image, /Dimensions)
IF N_Elements(xscale) EQ 0 THEN xscale = [0, s[0]]
IF N_Elements(xscale) NE 2 THEN $
    Message, 'XSCALE must be 2-element array', /NoName
IF N_Elements(yscale) EQ 0 THEN yscale = [0, s[1]]
IF N_Elements(yscale) NE 2 THEN $
    Message, 'YSCALE must be 2-element array', /NoName
```

Notice a new keyword for the *Size* command here: *Dimensions*. As opposed to the *N_Dimensions* keyword, which caused *Size* to return the number of dimensions of its argument, the *Dimensions* keyword causes *Size* to return a vector containing the size of each of its dimensions. In other words, with a 2D image array, I will get a two-element array containing the X size and Y size of the image, respectively.

Finally, I can check for the color keywords. Because I want to write device decomposed-state independent code, I am going to use the *GetColor* program to specify drawing colors. (*GetColor* is discussed in “Obtaining Device Independent Colors” on page 87.) *GetColor* “knows” the names of 16 drawing colors that I use frequently and can obtain those colors for me in a device independent way. (You can easily add more colors to *GetColor*.) I check the keywords like this:

```
IF N_Elements(dataColorName) EQ 0 THEN $
    dataColorName = "Red"
IF N_Elements(axisColorName) EQ 0 THEN $
    axisColorName = "Navy"
```

```
IF N_Elements(backcolorName) EQ 0 THEN $
    backcolorName = "White"
```

I could check to be sure the color names passed into the program are string variables, but I know *GetColor* is going to do that anyway. And I know that it is going to return to the caller of the program that called it. Thus, I will be able to catch that error in my error handler when it comes back from *GetColor*. Thus, there is no need to check for possible errors here.

Note that I make the background color white by default. This is certainly not necessary, and more often than not I like to have a nice charcoal or gray background color. I choose white here because it is sometimes easier to visualize what the results are going to look like when I make a PostScript file from this program. Recall that you can have any background color you like in PostScript, as long as that color is white. (This problem is discussed in “Problem: PostScript Devices Use Background and Plotting Colors Differently” on page 189.)

The most important thing is that I need some way to *change* the drawing colors, because I almost certainly *will* have to change them when I send the output to a PostScript printer. By making the default colors suitable for printing on a PostScript printer now, I won’t have to worry about resetting colors. I’ll just call the *HistoImage* program to draw the graphics in its default colors when I want to make a PostScript file.

If the user doesn’t supply a color table index number, I choose color table 4.

```
IF N_Elements(colortable) EQ 0 THEN colortable = 4
colortable = 0 > colortable < 40
```

Note that there are only 41 color tables supplied with IDL (although users could have modified this number, certainly). Here you see a bit of checking to force the *colortable* value into a number between 0 and 40. The IDL *greater than* and *less than* operators are used in a left to right fashion to force the value into the correct range of numbers. (In other words, 0 is compared to *colortable* and the largest value is returned. Then that value is compared to 40 and the smallest of those two values is returned into the *colortable* variable.) This kind of proactive error checking can avoid problems later on.

Next, I’ll supply the *ImageColors* keyword with a value. Note that I am going to use three drawing colors in this program. I prefer to load my drawing colors at the top of the color table, although other people prefer to load them at the bottom of the color table. It doesn’t matter much where you load them, as long as you know what you are doing when you manipulate the color table. But while I like to load drawing colors at the top of the color table, I don’t like to use the top index number of the color table.

The reason I don’t is that very often this index is used for the *!P.Color* system variable. And many, many programs are written assuming this color is going to be either white or black. I don’t like to break these programs, if I can help it. So I leave this index alone. I load my three drawing colors starting from the fourth index from the top. This means that the color indices I have for the image display ranges from 0 to the fifth index from the top of the color table. This is *!D.Table_Size*-4 total colors. This is what I assign to the *ImageColors* keyword value in the program.

```
imagecolors = !D.Table_Size-4
```

Note that I don’t have to check this output keyword. If the user wants the value he or she can get it back from the keyword. If they don’t want the value, fine. It didn’t cost me much of anything to assign it to a variable. (And I’ll need the value later in the program anyway.) There is no need to use something like *Arg_Present* in this case:

```
IF Arg_Present(imagecolors) THEN $
```

```
imagecolors = !D.Table_Size-4
```

This is overkill and results in the *imagecolors* variable only being defined if the user passed in a variable reference with the keyword. The simpler construction is much easier to type and I think makes the program easier to read, too.

Remember that the *imagecolors* variable is designed to help someone outside the program control the image colors. I don't have any need for that right now, but I may later and I want to be prepared for the eventuality.

Loading the Program Colors

The three drawing colors are going to be loaded starting at the fourth index from the top of the color table. I always use *!D.Table_Size* to indicate the size of the color table. Then *!D.Table_Size-1* is the top index in the color table. The code looks like this:

```
axisColor = GetColor(axisColorName, !D.Table_Size-2)
dataColor = GetColor(dataColorName, !D.Table_Size-3)
backColor = GetColor(backcolorName, !D.Table_Size-4)
```

The variables *backColor*, *dataColor*, and *axisColor* now contain either the correct color index number for loading the proper color (if device decomposition is off or if this is an 8-bit device), or a 24-bit value that can be decomposed into the proper color (if device decomposition is on). In any case, I don't have to worry at all about the current color decomposition state when I draw the graphics with these colors. The correct colors will appear almost automatically.

Next I load the colors for the image data. I only do this if the *NoLoadCT* keyword variable is *not* set.

```
IF NOT Keyword_Set(noloadct) THEN $
    LoadCT, colortable, NColors=imagecolors, /Silent
```

Note that I use the *Silent* keyword to the *LoadCT* command. I really don't like the informational message *LoadCT* prints in the command log window every time it loads a color table. This keyword suppresses the message.

Note, too, that I restrict, with the *NColors* keyword, the number of colors loaded to just those indices used by the image. I want to be careful not to overwrite the drawing colors I just loaded. Only color table indices 0 through *!D.Table_Size-5* will be loaded by this command.

Preparing to Draw the Graphics

The next step in writing the *HistoImage* program is to prepare to draw the graphics. I have three separate items to draw, the histogram plot, the color bar, and the image itself.

Calculating Graphic Positions in the Window

So the first thing I do is calculate the positions in the window where I want these items to go. These positions will be four-element arrays containing normalized window coordinates of the type that can be used with the *Position* keyword on most graphics commands. (See "Positioning Graphic Output in the Display Window" on page 44 for details of how this is done.) Normalized coordinates are used so the graphics will go into a window of any size. The code will look like this:

```
histoPos = [0.15, 0.675, 0.95, 0.95]
colorbarPos = [0.15, 0.500, 0.95, 0.55]
imagePos = [0.15, 0.100, 0.95, 0.40]
```

Changing Character Size According To Window Size

One of the requirements of this program is that it can display its graphics in resizeable graphics windows. Or, put another way, that it can display graphics in a window of any size. In other words, if the window is big the histogram plot should be big and the image display should be equally big. If the window is small, the plot and image display should be small, etc. This is accomplished, obviously, by the position of the various components in the window in normalized coordinates, as described above.

But often this thinking is not carried over to the annotation of the graphic displays as well. Most programmers will use a character size of 1 and call it good. But I would like to see a big character size used in big windows and a small character size used in small windows.

There is a *CharSize* keyword that can be used with graphics commands to change the character size, but how can this be done in a way that is consistent with the size of the window? The answer, like many of the answers you have discovered so far, is to express the character size in normalized units. Unfortunately, this is impossible in IDL. Character size is *always* expressed in character units. But even so...there must be a way!

Actually, there is. It turns out that by using the *XYOutS* command you can get the *width* of a text string in normalized units. And if you use a *negative* value with the *CharSize* keyword, then *XYOutS* doesn't actually write the string to the display window, it just calculates the width of the string. For example, suppose you wanted to know the width of the text string "A Sample String". You could type this:

```
IDL> thisSize = -1
IDL> XYOutS, 0.5, 0.5, 'A Sample String', /Normal, $
      Width=thisWidth, Charsize=thisSize
```

Now, suppose you compared the value of *thisWidth* with some target width. Say, for example, that the target width was 0.25. Another way of saying this is that the string should be wide enough to extend across 25 percent of the display window. If the target width was larger than the value of *thisWidth* you could increase the character size by some small amount, and test it again, and so on until the character size was appropriate to give you the text width you wanted. The code might look like this:

```
IDL> IF thisWidth LT 0.25 THEN thisSize = thisSize + 0.01
IDL> XYOutS, 0.5, 0.5, 'A Sample String', /Normal, $
      Width=thisWidth, Charsize=thisSize
```

Eventually, *thisWidth* would be within some small delta value of the target width. A similar algorithm can be employed if *thisWidth* is larger than the target width.

This kind of algorithm has already been developed for you in the form of the program *Str_Size* that you downloaded with the program files to use in this book. The parameters for *Str_Size* are a string and a target width, in normalized coordinates. I've found that the string "A Sample String" and a target width of 0.20 produces nicely sized characters on most plots in the graphics windows I typically use.

```
IDL> thisSize = Str_Size('A Sample String', 0.20)
```

This is a character size of approximately 1.4 for a normal sized window on a Windows machine, for example.

In this program, I will write the font character size selection code like this:

```
thisCharSize = Str_Size('A Sample String', 0.20)
```

Calculating the Image Histogram

The next step is to calculate the image histogram. (Recall that a histogram is simply a count of the number of entities in each bin of the histogram. Normally, we take this to mean the number of pixels with each image value.) I simply use the *Histogram* command, passing it the bin size I want to use, like this:

```
histdata = Histogram(image, Binsize=binsize, $  
Min=Min(image), Max=Max(image))
```

Notice I set the *Min* and *Max* keywords explicitly for the *Histogram* function to the minimum and maximum value of the image. The IDL documentation claims this is what the *Histogram* function does by default, but I have not found this to be the case. Rather, I find that it assumes a minimum value of 0 and a maximum value of 255 for byte images, no matter what the actual data values in the image.

Drawing the Graphics

There are three graphical elements I want to draw: a histogram plot, a color bar for indicating image values, and the image itself.

Drawing the Histogram Plot

IDL gives me several choices for drawing the histogram plot, but I find I don't like the two most obvious ones. Let me explain what I mean with a simple example you can type at the IDL command line. Suppose I have five bins of data, each bin being 5 units in size, and a vector that tells me how many "items" are in each bin. The *data* and *bins* vectors could be created like this:

```
IDL> data = [ 4, 6, 3, 8, 2 ]  
IDL> bins = [ 0, 5, 10, 15, 20 ]
```

The most obvious approach is to simply plot the data with the *Plot* command, like this:

```
IDL> Plot, bins, data, YRange=[0,10]
```

The result, which doesn't look much like our definition of a "histogram" plot, is illustrated in Figure 95.

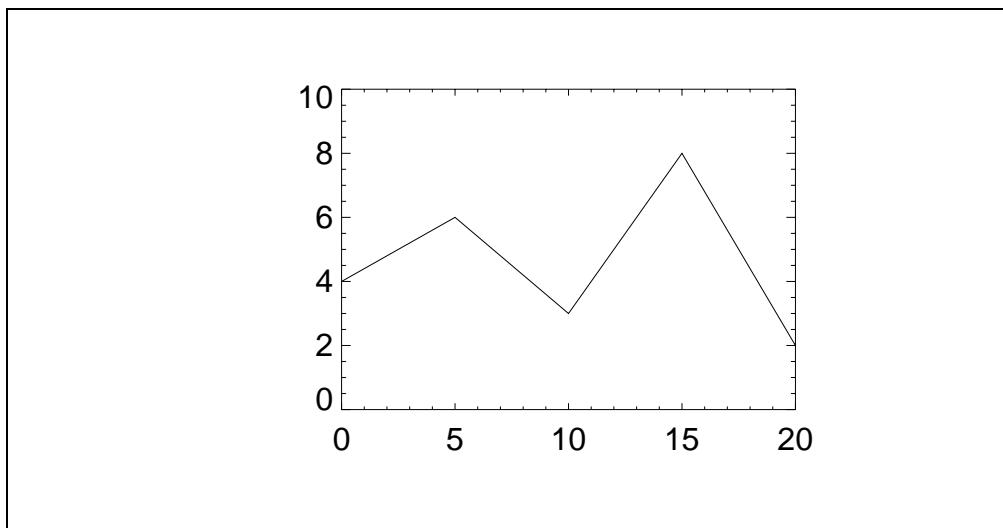


Figure 95: A very simple plot of a histogram function. Note that it doesn't look much like a histogram plot.

The second obvious approach is to set the *PSym* keyword to 10, which will result in the “stair-step” kind of plot we expect from a histogram plot. I can try this:

```
IDL> Plot, bins, data, YRange=[0,10], PSym=10
```

The results look better, as shown in Figure 96, but they are still not right. In particular, the bins are represented incorrectly. The first bin goes from 0 to 5, the second bin from 5 to 10, and so on. But in the illustration, the first bin appears to go from 0 to 2.5, and the second bin appears to go from 2.5 to 7.5, and so on. It appears as though each bin is half a bin size off.

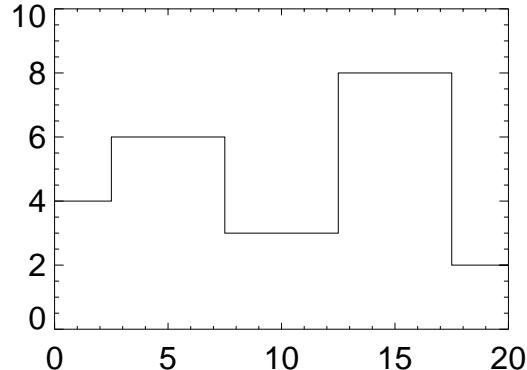


Figure 96: This plot looks more like a histogram plot, but it is still not right. Notice that the bins seem to be half a bin size off.

I could try to fix the problem by adding a half bin size to each of the bins, like this:

```
IDL> Plot, bins + 2.5, data, YRange=[0,10], PSym=10
```

You see the results in Figure 97. Again, this is close to being correct, but I have problems at either end of the plot, where the lines should extend to the end of the plot window. To really draw this plot correctly, I should duplicate the first and last values of the histogram data and the bin values.

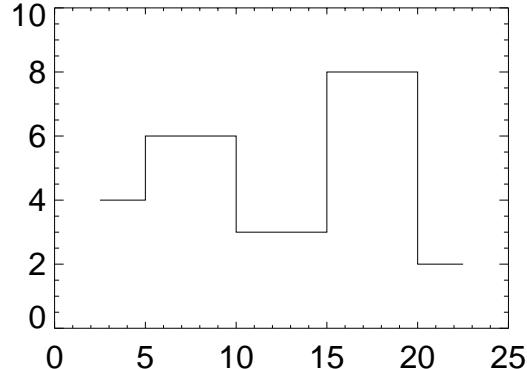


Figure 97: This plot has had half a bin size added to the bin values. It is accurate, but the plot lines do not extend to the ends of the plot window.

For example, I can do something like this:

```
IDL> Plot, [bins[0], bins + 2.5, bins[4] + 2.5 * 2], $  
[data[0], data, data[4]], YRange=[0,10], PSym=10
```

Finally, I get the kind of histogram plot I expect, as illustrated in Figure 98.

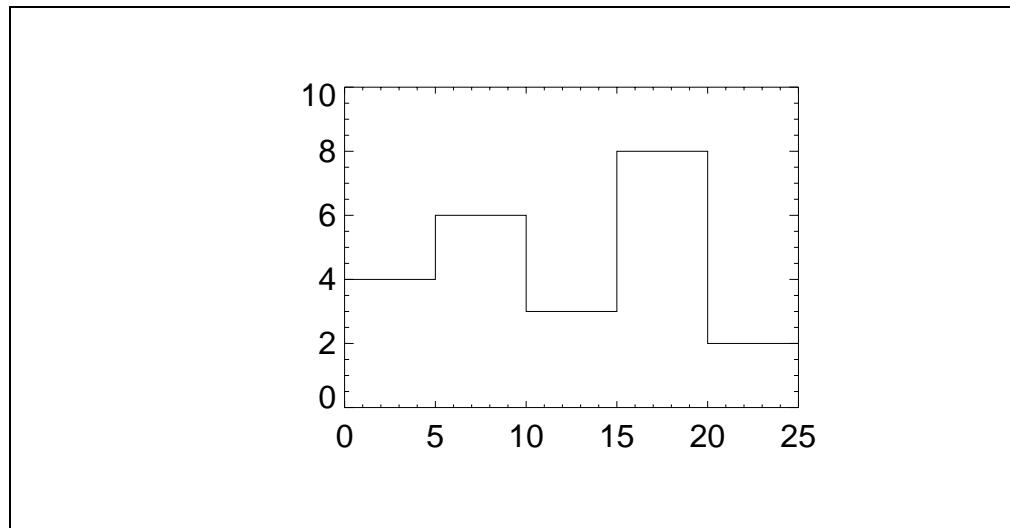


Figure 98: The histogram plot I expected to get. It would be easy enough to add vertical lines with the *PlotS* command to create histogram boxes for each bin.

This is exactly the kind of approach I am going to take in the *HistoImage* program. The code to fudge the *bins* and *histodata* vectors for correct plotting looks like this:

```
npts = N_Elements(histodata)  
halfbinsize = binsize / 2.0  
bins = Findgen(N_Elements(histodata)) * binsize + Min(image)  
binsToPlot = [bins[0], bins + halfbinsize, $  
             bins[npts-1] + binsize]  
histodataToPlot = [histodata[0], histodata, histodata[npts-1]]  
xrange = [Min(binsToPlot), Max(binsToPlot)]
```

The code to draw the histogram plot in the axes color, followed by the histogram data drawn in the data color will look like this:

```
Plot, binsToPlot, histodataToPlot, $  
      Background=backColor, $  
      CharSize=thisCharSize, $  
      Color=axisColor, $  
      Max_Value=max_value, $  
      NoData=1, $  
      Position=histoPos, $  
      Title='Image Histogram', $  
      XRange=xrange, $  
      XStyle=1, $  
      XTickFormat='(I6)', $  
      XTitle='Image Value', $  
      YMinor=1, $  
      YRange=[0,max_value], $  
      YStyle=1, $  
      YTickFormat='(I6)', $  
      YTitle='Pixel Density', $
```

```

    _Extra=extra
OPLOT, binsToPlot, histdataToPlot, PSym=10, Color=dataColor
FOR j=1L,N_Elements(bins)-2 DO BEGIN
    PlotS, Color=dataColor, [bins[j], bins[j]], $
        [!Y.CRange[0], histdata[j] < max_value]
ENDFOR

```

Notice that I use the *PlotS* command to draw vertical lines at each bin boundary. This gives the histogram a box look that I like better than just using the histogram plotting symbol (*PSym*=10) with the *Plot* command.



I find that by putting the keywords in alphabetical order when I use a command that requires setting a number of keywords a useful style. It makes it much easier to see which keywords I am explicitly setting. This saves time and effort if I have to add or delete a keyword later in program development.

Drawing the Color Bar

Drawing the color bar is easy. I simply use the *Colorbar* program you downloaded to use with this book. Like *TVImage* (see “An Alternative Image Display Command” on page 62), the *Colorbar* program is device decomposition independent and can be used in any IDL supported graphics device. The commands looks like this:

```

cbarRange = [Min(binsToPlot), Max(binsToPlot)]
Colorbar, $
    CharSize=thisCharSize, $
    Color=axisColor, $
    Divisions=0, $
    NColors=imageColors, $
    Position=colorbarPos, $
    Range=cbarRange, $
    XTickLen=-0.2, $
    _Extra=extra

```

Notice that the color bar is restricted to the same number of colors as the image, and that the range of colors is taken from the fudged *binsToPlot* variable. By setting the *XTickLen* keyword to a negative value, I produce outward facing tick marks. Setting the *Divisions* keyword to 0 allows the program to choose annotation divisions in the same manner as the *Plot* command. This will make the *Colorbar* annotation identical to the histogram plot annotations directly above it and will reinforce the purpose of the annotations.

Drawing the Image Plot

All that is left to do is draw the image, with its axes around it. I’ll use the *TVImage* command to display the image (see “An Alternative Image Display Command” on page 62) and the *Plot* command to draw the axes, using the values given by the *XScale* and *YScale* keywords as the range of the plot. The final code will look like this:

```

TVImage, ByteScl(image, Top=imageColors-1), $
    Position=imagePos, _Extra=extra

PLOT, xscale, yscale, $
    CharSize=thisCharSize, $
    Color=axisColor, $
    NoData=1, $
    NoErase=1, $
    Position=imagePos, $
    XStyle=1, $
    XTickLen=-0.025, $

```

```
YStyle=1, $  
YTicklen=-0.025, $  
_Extra=extra  
  
END
```

Notice that I scale the image data into the number of image colors and that I position both the image and the axes about the image with the *imagePos* variable.

Working Around a Printer Device Bug

There is a small *Printer* device bug in versions of IDL up through IDL 5.3.1 (the official version at the time this is written) that will cause a problem when this code is sent directly to a PostScript printer. (See “Loading Colors in the Printer Device” on page 204 for additional information.) It turns out that when a single color is loaded into the color table (in this program that is done with the *GetColor* command) at any index at all, then that same color is used to display any image pixel having a value of 0. (This is a *strange* bug!). For example, if you type these commands, and your default printer is a PostScript printer (PCL printers appear to be unaffected), then you might see output that looks like the illustration in Figure 99.

```
IDL> thisDevice = !D.Name  
IDL> Set_Plot, 'PRINTER'  
IDL> LoadCT, 0, NColors=!D.Table_Size-4  
IDL> HistoImage, /NoLoadCT  
IDL> Device, /Close_Document  
IDL> Set_Plot, thisDevice
```

The work-around for this bug, which will make the code completely device independent, is to simply get and re-load the color table vectors after the last single color is loaded into the color table. In the *HistoImage* code, find this line:

```
IF NOT Keyword_Set(noloadct) THEN $  
    LoadCT, colortable, NColors=imagecolors, /Silent
```

The line above should be replaced with this code:

```
IF NOT Keyword_Set(noloadct) THEN BEGIN  
    LoadCT, colortable, NColors=imagecolors, /Silent  
ENDIF ELSE BEGIN  
    IF !D.NAME EQ 'PRINTER' THEN BEGIN  
        TVLCT, r, g, b, /Get  
        TVLCT, r, g, b  
    ENDIF  
ENDIFELSE
```

This will cause the correct colors to be loaded when the program is sent to the *Printer* device.

Compiling and Testing the Program

To compile and test the program, type this:

```
IDL> .Compile histoimage  
IDL> HistoImage
```

If the program doesn’t compile, or if you have errors when you run it, delete the program from your display with the mouse, type *RECALL* at the IDL command line, and fix the errors. Don’t forget to re-compile the program before you try to run it again.

Try running the program with a different image:

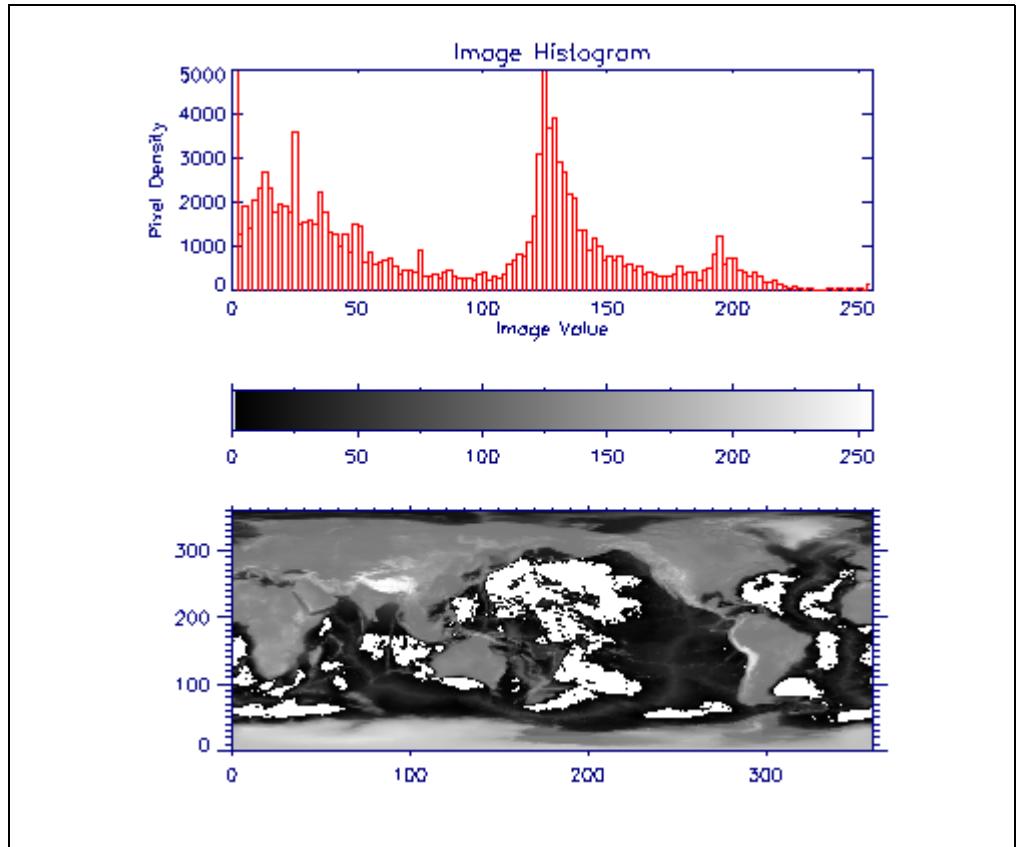


Figure 99: A Printer device bug that results in all pixels with value 0 showing up in the last single color loaded in the color table (the background color, in this case). The work-around is to get and re-load the color table vectors after loading a single color.

```
IDL> image = LoadData(5)
IDL> HistoImage, image
```

Try setting some of the keywords. For example, try some of the color keywords:

```
IDL> HistoImage, image, $
      AxisColorName='beige', $
      BackColorName='gray', $
      ColorTable=33, $
      DataColorName='yellow'
```

Try putting different scales on the image:

```
IDL> HistoImage, XScale=[0,1], YScale=[-1,1]
```

What about setting keywords that are not defined for *HistoImage*, but will get picked up by the keyword inheritance mechanism? Try, for example, setting the *Keep_Aspect_Ratio* keyword of the *TVImage* command and the *Divisions* keyword of the *Colorbar* command, like this:

```
IDL> HistoImage, /Keep_Aspect_Ratio, Divisions=8
```

Reviewing the HistoImage Program's Advantages

Let's review some of the *HistoImage* program's advantages. And I want to point out a few that may not be obvious to you even yet. First of all, the program has been written in such a way that it doesn't matter whether you are on an 8-bit display or a 24-bit

display, the program will work identically. Nor will it matter whether you have color decomposition on or off if you are on a 24-bit display. This is because we have used color-aware programs such as *GetColor* and *TVImage* to load drawing colors and display the image.

We have written the *HistoImage* program with an *_Extra* keyword defined for it. This allows us to pass “extra” keywords into the *Plot*, *Colorbar*, and *TVImage* commands inside the program. But it also does something far more useful. It allows us to write a program that can automatically re-display the graphic on 24-bit displays when we change color tables. (See “Automatic Updating of Graphic Displays When Color Tables are Loaded” on page 66 for additional information.)

For example, open a text editor and create this simple file, which you can name *histoimage_redisplay.pro*. (This program is one of the programs you downloaded to use with this book, if you prefer not to type it.)

```
PRO HistoImage_Redisplay, Image=image, _Extra=extra
IF N_Elements(image) EQ 0 THEN image = LoadData(7)
HistoImage, image, /NoLoadCT, _Extra=extra
END
```

This program will allow us to change the color table associated with *HistoImage* and see the effects immediately. (This will only be necessary on 24-bit displays, remember. On 8-bit displays, the colors are updated automatically.) Notice that the *NoLoadCT* keyword is set. This is necessary, you recall, for an outside entity to control the colors. If this keyword were not set, *HistoImage* would always load its own color table rather than using the colors of the current color table.

First, call the program normally and find out how many image colors there are. The *ImageColors* output keyword is used for this purpose.

```
IDL> image = LoadData(13)
IDL> HistoImage, image, ColorTable=33, ImageColors=ncolors
```

Next, call *XColors* to load different color tables. (*XColors* is a program you downloaded to use with this book. I use it exclusively in place *XLoadCT*, which is supplied with IDL, for reasons you will learn about in the following sections of this chapter. It has many advantages to *XLoadCT*, one of which is that I think it has a more natural syntax for automatically updating graphical displays on a 24-bit device.)

```
IDL> XColors, NColors=ncolors, Image=image, $
      NotifyPro='HistoImage_Redisplay'
```

If you have both *XColors* and your open graphics window on the display so you can see them both, you will notice that as you select color tables from the list of color tables in *XColors*, that the colors are automatically updated in the display window.

Note that if you are running IDL on an 8-bit device, you need only call *XColors* like this:

```
IDL> XColors, NColors=ncolors
```

Now, close your *XColors* window if it is still on your display.

The *XColors* program will work with the *HistoImage_Redisplay* program no matter what keywords you use with *HistoImage*. For example, you can type this:

```
IDL> HistoImage, image, BackColorName='gray', $
      AxisColorName='yellow', ImageColors=ncolors
IDL> XColors, Image=image, NColors=ncolors, $
      BackColorName='gray', AxisColorName='yellow', $
      NotifyPro='HistoImage_Redisplay'
```

The HistoImage Program is Device Independent

We have seen that the *HistoImage* program is visual display depth independent and color decomposition independent, but what may not be immediately obvious is that it is also device independent. That is to say, it doesn't matter which graphical display device you have currently selected to display graphics, the *HistoImage* program will work correctly in that device. This includes such devices as the PostScript device (*PS*), the printer device (*PRINTER*), and the Z-graphics buffer device (*Z*).

For example, to print this on your default printer, you can type this:

```
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PRINTER', /Copy
IDL> Device, XSize=5, YSize=5, /Inches, XOffset=1.75, $
      YOffset=3.0
IDL> HistoImage, image
IDL> Device, /Close_Document
IDL> Set_Plot, thisDevice
```

Or, to create a PostScript file, you can type this:

```
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PS'
IDL> Device, XSize=5, YSize=5, /Inches, XOffset=1.75, $
      YOffset=3.0
IDL> HistoImage, image
IDL> Device, /Close_File
IDL> Set_Plot, thisDevice
```



Recall that I set the background color to be white by default just for the purpose of being able to send the graphic output to a printer or to the PostScript device. If it is not white on the display, you should change it to white before you send it to the printer or create a PostScript file. The color keywords make it easy to make this switch when we are using these types of graphics output devices.

What makes the *HistoImage* program device independent is the absence of commands that only work in a particular device. For example, you see no *Window* commands in this program, since a *Window* command is only appropriate on display devices and not, for example, in the PostScript device. The graphics have been written to go into any window that happens to be open. If the graphics device is a display device (i.e., *WIN*, *MAC* or *X*), and a window is not currently open, a default window will be opened when the first graphics command is executed.

If you did want to have a *Window* command in the program (and I almost never do, especially if I have plans to use the program in a widget program), then you should make sure the device supports windows. This can be done with the *Flags* field of the *!D* system variable. The flags field is a bit map and we want to see if the bit corresponding the value 256 is set (windows are supported by this device) or not (windows are not supported by this device). The code will look like this for a possible *Window* command:

```
IF (!D.Flags AND 256) NE 0 THEN Window
```

Another common command you don't see in this program is a *Device, Decomposed=0* command. Normally this command is required to display 2D images in color. We don't have to include it because the *Colorbar* and *TVImage* commands have been written with this intelligence built into them. However, we would have to use a command like this if we were going to use the *TV* command to display the color bar and image. In fact, we would probably have to use *several* commands to check the visual depth, the type of device, etc. For details, examine the code in either the *TVImage* or *Colorbar* programs. In the *Colorbar* program 12 lines of code precede the

TV command and another 12 lines follow it, just to make the *TV* command work properly on every device!

Using *HistoImage* in a “Smart” Resizeable Graphics Window

The *HistoImage* program also meets the graphics display criteria for being displayed with a resizeable graphics window program, named *FSC_Window*, which you downloaded to be used with this book. The *FSC_Window* program is a “smart” graphics window, in that it can resize its contents, create BMP, GIF, JPEG, PICT, PNG, TIFF, and PostScript files of its window contents, and send its contents directly to the printer. Plus, if your graphic display program has been written correctly, you can also change the colors in your program with a color table changing tool.

The five criteria *FSC_Window* imposes on a display program are these:

- 1. The program should be written as a procedure.
- 2. There should be no more than three positional arguments.
- 3. There can be an unlimited number of keyword arguments.
- 4. The program should be written so that the contents goes into any sized window.
- 5. There should be no device-specific commands in the program (e.g., a *Window* command).

Many graphics display commands meet this criteria. For example, the *Shade_Surf* command does. Type these commands:

```
IDL> peak = LoadData(2)
IDL> LoadCT, 22
IDL> FSC_Window, 'Shade_Surf', peak, CharSize=1.5
```

You see an example of the *FSC_Window* program in Figure 100. Notice the controls under the *File* button in the menu bar. You can grab the edge of the window with the mouse and resize the window. The window contents resize themselves to fit.

You can have as many *FSC_Window* programs running as you like. For example, let’s display an image in an *FSC_Window* program, but let’s also set the ability of the program to load different color tables. Type this:

```
IDL> image = BytScl(LoadData(7), Top=!D.Table_Size-1)
IDL> FSC_Window, 'TVImage', image, /WColors
```

Notice now there is a *Colors* menu item under the *File* menu bar button. Clicking this button will call up the *XColors* color changing tool. *XColors* is a program you downloaded to use with this book. One of its huge advantages is that it doesn’t use common blocks to store its color tables. Which means there can be multiple *XColors* programs on the display at once. Something that is impossible for *XLoadCT*, the IDL-supplied color table changing tool.

For example, let’s get the *HistoImage* program on the display too. Recall, though, that you do not want to use all of the colors in the color table. We reserved the top four colors for other things. And recall that if we want colors to be manipulated externally, we have to be sure *not* to load the color table in *HistoImage*. This is accomplished by setting the *NoLoadCT* keyword. Type this:

```
IDL> image = BytScl(LoadData(5), Top=!D.Table_Size-1)
IDL> FSC_Window, 'HistoImage', image, /NoLoadCT, $
    WColors=!D.Table_Size-4
```

Now you can get the color table tool from both programs on the display simultaneously. If you are running IDL on an 8-bit display, things look a bit chaotic, no doubt, since every time the color table is changed, all graphics output changes automatically.

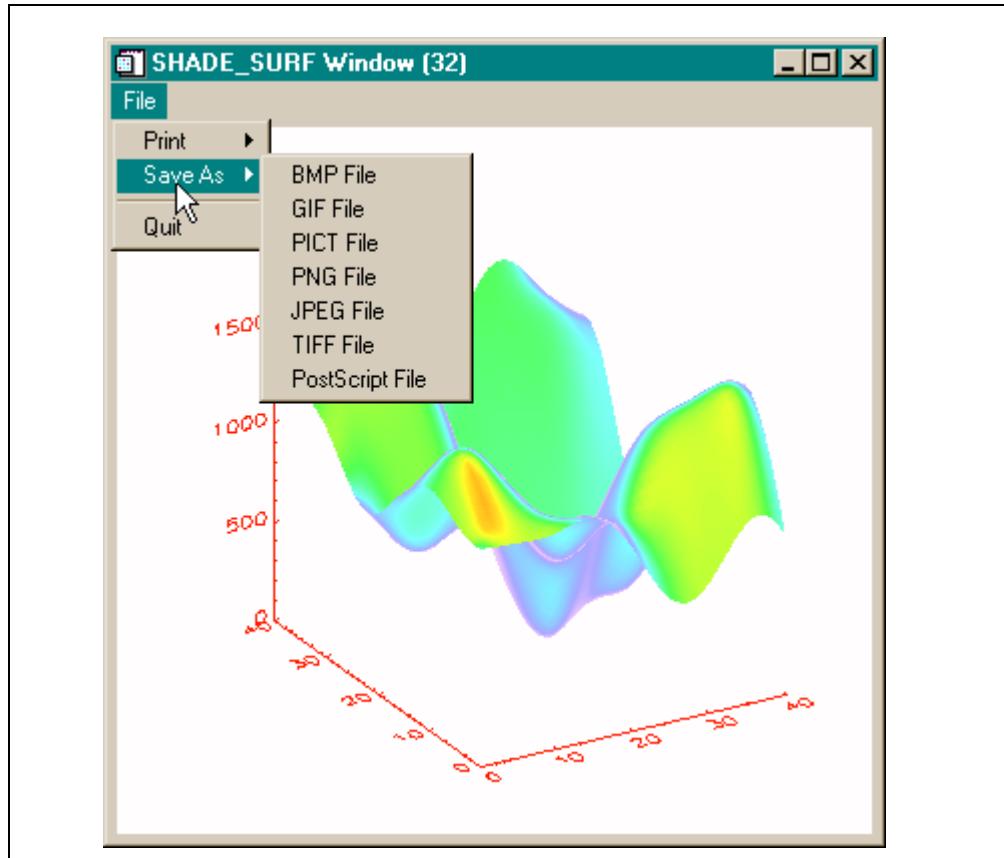


Figure 100: The *FSC_Window* resizable graphics window program with a *Shade_Surf* command. Notice the pull-down menu that allows you to print and save the window contents in a variety of file formats.

On a 24-bit displays, of course, this all happens independently. What you will notice on 8-bit displays, however, is that the color will be correct for the window that has the current keyboard focus. In other words, each window “knows” which colors are supposed to be loaded and loads them when it has the focus.



Note that on an 8-bit display, the *FSC_Window* program may not have a *Print* and *PostScript File* button as shown in Figure 100. When the display device does not have as many colors as the PostScript or printer devices, it is impossible to always get the colors correct for PostScript and printer output. Too many factors are involved and it depends too much on how the graphics command that is executed is written. For example, it is not possible to get correct PostScript output from our *HistoImage* program when it is running on an 8-bit display with color table loading turned off, as is the case currently.

You will learn more about how to fix these kinds of problems in the chapters to follow, but for now you should know that you can get both a *Print* button and a *PostScript File* button on your *FSC_Window* program on an 8-bit display, but you have to set them explicitly. For example, you could do this:

```
IDL> FSC_Window, 'TVIImage', LoadData(7), /WColors, $  
/WPostScript, /WPrint
```

But even this command would not print correctly or make a correct PostScript file on an 8-bit display. To make the output correct, the image data will have to be scaled correctly for both the display and for the PostScript and printer devices. The only way

this can be accomplished it to perform the scaling right in the *TVImage* command, like this:

```
IDL> image = LoadData(7)
IDL> FSC_Window, 'TVImage', $
      BytScl(image, Top=!D.Table_Size-1), /WColors, $
      /WPostScript, /WPrint
```

Chapter 10

◆ Discovering the Possibilities ◆◆◆



Writing a Widget Program

Chapter Overview

The purpose of this chapter is to demonstrate how to write a basic IDL program using the widget toolkit to provide a graphical user interface to the program. The name *widget* refers to a graphical element that the user interacts with to convey information into or out of a program. Buttons, slider bars, and fields where you type text are all examples of widgets. The widget toolkit is a set of IDL commands that together form a special application programming interface (API) for building programs with graphical user interfaces. Specifically, you will learn:

- The difference between a widget definition module and a widget event handler module
- How to write a widget definition module
- How to write a widget event handler module
- How to interpret and understand widget event structures
- How to create different types of widgets, including top-level base widgets, draw widgets, and pull-down menus
- How to pass information between widget program modules without using common blocks
- How to write a widget program with a resizeable graphics window
- How to add simple controls to a widget program
- How to change and protect colors in a widget program
- How to prevent memory leakage in a widget program

The program you write in this chapter, named *Histo_GUI*, will be a graphical user interface for the *HistoImage* program you wrote in the previous chapter. Initially, this program will simply be a resizeable graphics window for the *HistoImage* program. In the next chapter you will add more features and complexity to the program.

The Structure of Widget Programs

While many IDL programs (like the *HistoImage* program in the previous chapter) are written as a single program module, widget programs always contain at least two program modules. There is always a *widget definition module*, which is the program

module (either an IDL procedure or function) in which the widgets themselves are created or defined. And there is always at least one *event handler module*, which is the program module (either an IDL procedure or function) which processes or responds to widget *events*, the triggers or user interactions that drive the widget program. (See Figure 101, below.)

Even simple widget programs may have several event handler modules, each of which responds to a different kind of event or trigger. Program information often needs to be shared among these event handlers and the widget definition module. Much of the art of widget programming is learning how to pass program information between the various program modules that constitute a widget program or application.

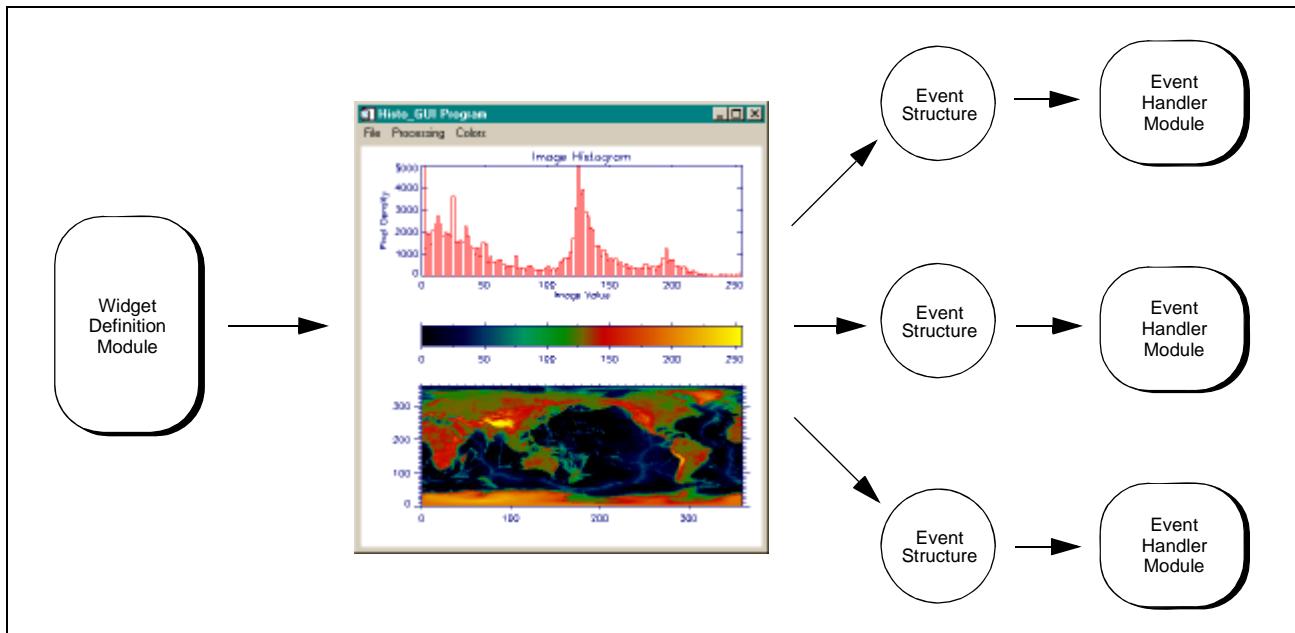


Figure 101: The flow of information in a widget program passes from the widget definition module, where the widgets are defined and the program is realized on the display, through event structures to the widget event handler modules. In general, code in the widget definition module is executed only once, whereas code in the event handler modules may be executed over and over again, each time there is an event appropriate for that event handler.

Widget programs are said to be *event driven*. This means that the program does not know in advance the order of program execution. Events are generated one at a time by users interacting with the graphical elements of the widget program, and event handler modules respond to each event, separately, in the order in which the events are generated. Each event is processed completely before the next event is processed. If the program generates events faster than they can be processed, the events get queued to await processing at the first opportunity.

In general, the interface or visual display of the widget program is generated in the widget definition module. And the “action” of a widget program is generated in the event handler modules. In fact, the code in the widget definition module of most widget programs is executed only once, while the code in the event handler module or modules is executed many, many times. In fact, the event handler code is executed each time there is an event generated that is appropriate for that event handler.

How Do Widget Programs Respond to Events?

One of the responsibilities of the widget definition module, in addition to creating the program's widget interface, is to set up what is often called the widget *event loop*. This is an unfortunate name because there is no actual loop anywhere. You might better think of the widget program as being in a state of attention. It is waiting, as it were, for something to happen to one of its widgets. But who—or what—is waiting?

As it happens, it is the window manager that is waiting. At the same time that the widget definition module sets up the event loop, it turns control of the program over to the window manager. (Both of these operations are done with a single *XManager* command, as you will see below.) The window manager is then responsible for knowing that some kind of *event* or user interaction has occurred in a program that it is responsible for managing.

Upon learning that an event has occurred, the window manager packages information about that event up into a packet that it sends to IDL. IDL takes the event information (in the *XManager* code) and re-packages it into what is called an *event structure*. Then IDL calls the appropriate event handler module for the widget that caused the event. In calling the event handler, IDL passes the event structure to the event handler as the event handler's one and only positional parameter. (See *Appendix A: Widget Event Structures* for a list of the different event structures that can be passed to event handlers from the widgets supplied in the IDL distribution and with this book.)

Although the specific fields in any event structure vary, depending upon which widget caused the event, every event structure is the same in that it must have three specific fields: *ID*, *Top*, and *Handler*. In fact, it is the presence of these three fields in an IDL structure variable that makes the structure an event structure as opposed to some other kind of structure. You will learn the importance of these three fields and how they can be used to your advantage in a moment.

Writing the Widget Definition Module

The first thing that is needed in a widget program is the *widget definition module*. There are four purposes of the widget definition module: (1) to create and define the widgets that together constitute the program's graphical user interface, (2) to realize the widget hierarchy (make the created widgets appear on the display), (3) to set up the program's event loop, and (4) to register the program with the window manager, which is then responsible for responding to program events. The widget definition module is an IDL program module like any other and can be either a procedure or a function. You define positional and keyword parameters for the program module in the normal way.

The program you will write here will be a wrapper for the *HistoImage* program from the previous chapter. That is to say, it will use the *HistoImage* program to display graphics. But it will have graphical user interface elements that will allow the user to control how the program appears in the display. This program will be named *Histo_GUI*. Open a new text editor window, save the file as *histo_gui.pro*, and type the code that follows in this chapter.

The Advantage of Mistakes



I much prefer that you type the program code in IDL as you work with this book. If you do, you will make many more mistakes and by that means learn more than you will by examining code I write for you. (Although, I suppose you won't have to be particularly sharped eyed to see mistakes there, too, if the past is any indication of the future.) Most learning comes from making mistakes and having to solve the problems

those mistakes pose for us. My teaching style is to let users make lots of mistakes. Sometimes I deliberately lead users into mistakes, so they get used to figuring things out. You cannot possibly be a good IDL programmer without this essential skill.

But I also realize we are all busy and sometimes we appreciate shortcuts. I don't believe there *are* any shortcuts when it comes to learning a programming language, but I am sympathetic none the less. To that end, I have prepared a series of files that represent the *Histo_GUI* program in various stages of development. These files are among the files you downloaded to use with this book. For example, the file that represents the *Histo_GUI* program as I describe it in this section is named *histo_gui.1.pro*. I have added the extra "1" extension so that if you type *Histo_GUI* at the IDL command line, you cannot mistakenly run a file I provided for you, rather than the one you are working on. To run this file (and the others like it), you must first compile the file explicitly:

```
IDL> .Compile histo_gui.1.pro
```

Then you can run it in the normal way:

```
IDL> Histo_GUI
```

If you consistently work on the file you named *histo_gui.pro*, then you can always switch back and forth from your file to mine by compiling whichever file you like and typing *Histo_GUI*.

Typing the Code

Since the program as you are writing it is a wrapper program that will pass its arguments to the *HistoImage* program, you will want to define the same positional arguments and most of the same keyword arguments that were defined for the *HistoImage* program. Certainly you will want to define those keyword arguments that set properties of the program that you may wish to control externally. For example, the drawing colors, the bin size, the maximum value for the histogram plot, etc. (See "Writing the Procedure Definition Statement" on page 240 for additional information.) The procedure definition line for the *Histo_Gui* program will look like this:

```
PRO Histo_GUI, $ ; The program name.  
    image, $ ; The image data.  
    AxisColorName=axisColorName, $ ; The axis color.  
    BackColorName=backcolorName, $ ; The background color.  
    Binsize=binsize, $ ; The histogram bin size.  
    ColorTable=colortable, $ ; The colortable index.  
    DataColorName=datacolorName, $ ; The data color.  
    _Extra=extra, $ ; Extra keywords.  
    Max_Value=max_value, $ ; The histogram max value.  
    Title=title, $ ; The program title.  
    XScale=xscale, $ ; The X image scale.  
    YScale=yscale ; The Y image scale.
```

The next item is usually some kind of error handler. (See "Writing the Error Handling Code" on page 242 for additional information.) You can write a simple error handler like this:

```
Catch, theError  
IF theError NE 0 THEN BEGIN  
    Catch, /Cancel  
    ok = Error_Message(Traceback=1)  
    RETURN  
ENDIF
```

As with all procedures and functions, the next step is to check for the presence of positional and keyword arguments, and define values for any that are not there. (See “Checking for Positional and Keyword Parameters” on page 242 for additional information.) The code will look remarkably similar to the code in *HistoImage*:

```

IF N_Elements(image) EQ 0 THEN image = LoadData(7)
ndim = Size(image, /N_Dimensions)
IF ndim NE 2 THEN $
    Message, '2D Image Variable Required.', /NoName
IF N_Elements(binsize) EQ 0 THEN BEGIN
    range = Max(image) - Min(image)
    binsize = 2.0 > (range / 128.0)
ENDIF
IF N_Elements(max_value) EQ 0 THEN max_value = 5000.0
IF N_Elements(title) EQ 0 THEN title = 'Histo_GUI Program'
s = Size(image, /Dimensions)
IF N_Elements(xscale) EQ 0 THEN xscale = [0, s[0]]
IF N_Elements(xscale) NE 2 THEN $
    Message, 'XSCALE must be 2-element array', /NoName
IF N_Elements(yscale) EQ 0 THEN yscale = [0, s[1]]
IF N_Elements(yscale) NE 2 THEN $
    Message, 'YSCALE must be 2-element array', /NoName
IF N_Elements(dataColorName) EQ 0 THEN $
    dataColorName = "Red"
IF N_Elements(axisColorName) EQ 0 THEN $
    axisColorName = "Navy"
IF N_Elements(backcolorName) EQ 0 THEN $
    backcolorName = "White"
IF N_Elements(colortable) EQ 0 THEN colortable = 4
colortable = 0 > colortable < 40
imagecolors = !D.Table_Size-4

```

The next step is to create the program’s widgets.

Defining and Creating the Program’s Widgets

Simple or basic widgets are created with the widget creation routines available in the IDL widget toolkit API. These routines are IDL functions that, in general, have a syntax like this:

```
widgetID = Widget_NAME(parentID)
```

where the *NAME* is the type of widget that can be created. Possible values are: *Base*, *Button*, *Draw*, *Droplist*, *Label*, *List*, *Slider*, *Table*, and *Text*. The *parentID* variable is the identifier of the widget’s *parent*, or widget directly above this widget in a widget hierarchy. (See Figure 102 for an example of a widget hierarchy.)

Widget hierarchies resemble family trees when diagrammed. Thus, we speak of widgets as if they have family relationships. We often talk about the *parent*, *child*, or *sibling* of a widget. What we are describing is how the widgets are connected or related to one another. It is the purpose of the widget definition module to create each widget and describe its relationship to other widgets in the program through the *parentID* parameter.



Note that widgets only have one parent. I’m not a biologist, so I don’t know how this is done, but I hear it is a fascinating story.

Every widget hierarchy has at its head a top-level base widget. A base widget is a container for holding other widgets. The top-level base widget is the container that

holds the other widgets that together constitute the graphical user interface of the program. It is, essentially, the program window that appears on the display. The top-level base widget is the only widget that can be created in a widget program without a *parentID* parameter. All other widgets require a single *parentID* parameter be specified in the widget creation function.

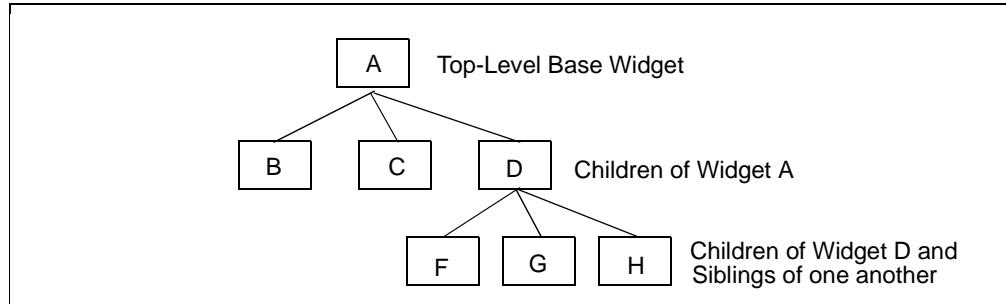


Figure 102: A diagram of a widget hierarchy. Widget A is a top-level base widget and is the container that holds all the other widgets. Widgets B, C, and D are children of widget A, who is their parent, and siblings of one another. Widgets F, G, and H are children of widget D and are siblings of one another, etc.

Notice that as functions, the widget creation routines always return a value. This *widgetID* value is the identifier of the particular widget that is created. It is guaranteed to be a unique long integer number. If you want an analogy from the real world, you can think of this number as the widget's social security or tax identifier number. It is a number that identifies this widget as different from all other widgets, even those with the same name. If you wish to work with this widget in the future (and you certainly will with many widgets), then you will need to keep track of this number in your program, as this is the way this particular widget will be identified by the program.

Creating the Top-Level Base Widget

Arguably, the most important widget you create in a widget program is the top-level base. The top-level base is the container that holds all the other widgets in the program. It is, in effect, the actual window that appears on your display, containing the program's widget interface. Although a top-level base widget does not require a *parentID* parameter in its widget creation function, it may have other properties set for it via keyword parameters to its widget creation function.

In this case, you would like to give the top-level base widget a title, you would like the children of this widget to be stacked up in a single column as they are created, you would like to make the top-level base resizable, and, finally, you would like this top-level base to have a menu bar similar to many of the programs you are used to seeing and using on a computer, with buttons to control program operation. The code to define such a top-level base will look like this. Type:

```
tlb = Widget_Base(Column=1, TLB_Size_Events=1, $  
Title=title, MBar=menubarID)
```

Notice that while the *Column*, *Title*, and *TLB_Size_Events* keywords are all input keywords (i.e., information is going *into* the *Widget_Base* routine), that the *MBar* keyword is an *output* keyword (i.e., information is coming *back* from the *Widget_Base* routine). There is absolutely no difference in syntax between an input or output keyword, so you will have to learn the difference from context and from reading the documentation for the routine. (For additional information about keyword parameters, see “Defining a Keyword Parameter” on page 213.)

In this case, the *menubaseID* variable is undefined until the *Widget_Base* command puts a value into it. The *menubarID* variable makes it possible to put buttons into the menu bar of the top-level base, since the *menubarID* variable can be used as the parent identifier for the menu bar buttons.

Creating Buttons for the Menu Bar

Next, define a *File* menubar button, with a *Quit* button in its pull-down menu. The code will look like this:

```
fileID = Widget_Button(menubarID, Value='File', /Menu)
quitID = Widget_Button(fileID, Value='Quit', $
Event_Pro='Histo_GUI_Quit')
```

Notice how the *menubarID* variable is used as the parent of the *File* button. The *File* button, in return, is the parent of the *Quit* button (through its *fileID* identifier). Button widgets cannot normally be the parents of other button widgets. It is the *Menu* keyword which is set for the *File* button that makes this possible. The *File* button with the *Menu* keyword set becomes a pull-down menu button, which *can* have children.



Note that the *Menu* keyword is not strictly required in the button creation function for the *File* button, although it does no harm to set it in this case. Any button that is a child of a menu bar (via the *MBar* keyword in a top-level base creation routine) is, by definition, a menu bar button and the start of a pull-down menu. Menu bar buttons *never* generate events on their own. They only expose the buttons that are their children. That is, the buttons that are associated with their pull-down menu.

The values of these two buttons are the text strings (“File” and “Quit”) that are written on the buttons. In general, if you deal with the values of buttons in event handlers, you should remember that these strings are case sensitive.

Notice the *Event_Pro* keyword in the *Quit* button creation routine. I am using this keyword to assign the name of an event handler module to this *Quit* button. Every widget you create can have its own event handler module assigned to it, although most programs are not written this way. Some programmers write a single event handler module that handles all of the events in the widget program. Other programmers (and I am among them) find it easier to maintain and extend programs if there are multiple event handler modules in a program. I know for a fact that it is easier for people just learning about event handlers to have separate event handler modules in their programs. It makes the program *much* easier to understand. Note that the event handler module will need a unique name. Typically, I use the name of the widget definition module (*Histo_GUI*, in this case) with an extension that tells me something about the purpose of the event handler. This makes it much easier to read my code later on.

Event handlers may either be procedures or functions. The *Event_Pro* keyword attaches an event handler procedure. If I wanted the event handler to be a function, I would use the *Event_Func* keyword to assign it.



Note that you should *never* assign an event handler to the top-level base using these keywords. You will see in just a minute that the event handler for the top-level base must always be a procedure and it must always be assigned with the *Event_Handler* keyword to the *XManager* command.

Creating the Graphics Window for the Program

The next widget you should create is the program’s graphics window. This will be a draw widget. A draw widget is similar to a normal IDL graphics window, although not *exactly* the same. For example, you do not want to ever use a *Cursor* command in a draw widget window, although you can use a *Cursor* command in a regular IDL graphics window.

(Using the *Cursor* command in draw widget windows may cause strange things to happen in widget programs. Things which would appear to have nothing whatsoever to do with a *Cursor* command. As you will see, using a *Cursor* command in a draw widget window is completely unnecessary, since the draw widget event structure itself already contains the information you seek with the *Cursor* command.)

To create the draw widget and make it a child of the top-level base type:

```
drawID = Widget_Draw(tlb, XSize=400, YSize=400)
```

Notice that the sizes of the draw widget are in pixel units, just like the sizes of a regular IDL graphics window.

Realizing the Widgets on the Display

As widgets are created with the widget creation routines, they exist only in IDL's memory. They do not appear on the display. To make the widgets appear on the display they must be *realized*. In practice, the top-level base is realized and that realizes all of the widgets that belong to that widget hierarchy.

Realizing a widget is a way of interacting with it. In IDL, all widget interaction takes place with the *Widget_Control* command. The command requires the identifier of the widget you wish to interact with. Specific interactions are put into effect by means of specific keywords. The *Widget_Control* command has over 60 different keywords to interact with widgets in various ways.

To realize the entire widget hierarchy, you can type this command:

```
Widget_Control, tlb, /Realize
```

Making the Draw Widget the Current Graphics Window

Unlike a regular IDL graphics windows, draw widgets do not become the current graphics window when they are created. In fact, you must specifically set the draw widget window to be the active or current graphics window before you can draw graphics into it. You do this, as with a regular IDL graphics window, by using the *WSet* command and the window's graphics index number. But you must remember that the draw widget's identifier (*drawID* above) is *not* the draw widget's graphics index number.

In fact, draw widgets do not get assigned a graphics window index number until *after* they have been realized. And then the graphics window index number of the draw widget is the draw widget's *value*. This is very different from the draw widget's identifier (i.e., its social security or tax ID number). Be sure you don't confuse them.

Since the draw widget, as one of the children of the top-level base, was realized by the command above, it has been assigned a graphics window index number by IDL. Get that number and make the draw widget the current graphics window, by typing this code into the program file:

```
Widget_Control, drawID, Get_Value=wid  
WSet, wid
```

Notice that the *Get_Value* keyword is an output keyword. The draw widget's graphics window index number is being put into the variable *wid*.

Displaying the Graphics in the Draw Widget Window

The graphics are easily displayed in this draw widget window by simply calling the *HistoImage* program you wrote earlier, passing it the parameters you collected (or created) here. (If you didn't write the *HistoImage* program in the last chapter you can

find it among the files you downloaded to use with this book.) You see now the advantages of writing the *HistoImage* program the way you did. Since you wrote the program in such a way that it can go into the current graphics window without regard for the size of the window, it can be called like this:

```
HistoImage, image, $
AxisColorName=axisColorName, $
BackColorName=backcolorName, $
Binsize=binsize, $
ColorTable=colortable, $
DataColorName=datacolorName, $
_Extra=extra, $
ImageColors=imagecolors, $
Max_Value=max_value, $
XScale=xscale, $
YScale=yscale
```

Note that the variable *imagecolors* is an output variable that tells us the number of colors associated with the image data. This is important in this widget program because we eventually want to control those colors with a color table changing tool. We will add that capability to the program in the next chapter.

Storing Information Required to Run the Program

Almost every widget program needs information to run the program successfully. This might be widget identification numbers, program data, graphics window index numbers, and other information or variables to make the program run properly. Quite often this information is created or available in one program module, but it is needed in another program module. For example, most widget identifiers are created in the widget definition module, but they are used in the event handler modules, where the program action occurs.

Much of the art of widget programming comes from knowing how to pass this information from one program module to another. There are several ways to do it. Many programmers, for example, use common blocks to make this kind of information available to program modules.

But common blocks almost by definition make your widget programs less useful, since you can have only one instance of the program running at any one time without confusing the common blocks. Consider, for example, two separate instances of the same image processing program that stored its image data in a common block named *Data*. If these two instances were loaded initially with different image data sets, the first program could never process its own image data. Both programs could only process the last image to be loaded into the common block. For this reason and others, I like to write widget programs that don't use common blocks at all.

To avoid common blocks, I use what I call an *info structure* to store essential program information. This is an anonymous IDL structure variable that holds any information I need in my widget program. I've learned from long experience that when I create the *info* structure that it is best to give the fields of the structure the same names as the variables that go into them. Although this convention is certainly not required, it results in less confusion. I also strive to give my variables the same names in my widget programs. This makes it easy for me to know what I am working with when I see the name in program code.



Note that the *info* structure is an anonymous structure because I will inevitably want to add fields to it as I work with the program and discover information that I have left out that I need. It will be impossible to extend this structure if it is a named structure without either exiting IDL or resetting my IDL session with the *.Reset_Session* executive

command (in IDL 5.3 and higher). Thus, an anonymous structure is much easier to work with.

In this program, I am going to need several pieces of information. Certainly I will need the image data and other information associated with calling the *HistoImage* program correctly. In addition, I will need the draw widget identifier (so I can resize the draw widget when I need to) and the window index number of the draw widget (so I can make it the active graphics window before I draw into it). You might want to define the *info* structure like this:

```
info = { image:image, $  
        axisColorName:axisColorName, $  
        backColorName:backcolorName, $  
        binsize:binsize, $  
        dataColorName:datacolorName, $  
        imageColors:imagecolors, $  
        max_value:max_value, $  
        title:title, $  
        xscale:xscale, $  
        yscale:yscale, $  
        extra:extra, $  
        drawID:drawID, $  
        wid:wid $  
    }
```

But there are two problems with this.

The first and most obvious problem will probably occur when you try to run this program the first time. This line will have an error in it. The error will be that the *extra* variable is undefined.

Recall that the *extra* variable is not really being created in your program. IDL itself is creating it if extra variables are passed into the program via the keyword inheritance mechanism. (See “Passing Undefined Keywords by Keyword Inheritance” on page 216 for more information.) You could, obviously, check to be sure it is defined in the program, but what if it isn’t? What value should you assign to it?

Humm. Good question. Let’s forget this for a moment and come back to it after we consider the second problem.

The second problem isn’t really a problem *now*, but it could become a problem later. It concerns the *image* variable. There is no problem putting the *image* variable into this *info* structure the way I just suggested. IDL will examine the variable and allocate enough memory space in the structure to accommodate the image. But what if you decide you want to load a new image into the program? (Have you ever seen a program with a graphical user interface that *didn’t* allow you to load new data?) What if that image is a different size than the old one? It will either not take up the same amount of room in the structure (if it is smaller than the current image), or it will not fit into the structure (if it is larger than the current image) and will cause an error. In either case, you have problems.

As it turns out, both of these problems can be solved by using pointer variables.

Using Pointer Variables

Pointer variables, or pointers, in IDL should not be confused with, say, pointers in the C language, which are actually references to machine memory locations. IDL wouldn’t allow you to use something as dangerous as that. Research Systems would have to increase their technical support staff by a factor of five just to keep up with the increased demand on their time.

No, IDL pointers are almost as *useful* as C pointers, but they are a lot less lethal. They are actually a special type of variable called a *heap variable*. (Objects are another type of heap variable.) And don't confuse the word *heap* with, for example, the heap in the C programming language. It is just an area of global memory where these special types of variables can be stored.

The huge advantage of heap variables is that they are in some sense global in scope. Well, global in the sense that, say, common blocks are global in scope. That is to say, if you have the "address" of the pointer data, you can access the data. The "address" is the pointer variable reference itself, a lightweight (only 4 bytes) token that can easily be passed around among program modules.

Pointers are perhaps easier to understand by example. Suppose we want to store an image at a pointer location. We create the pointer and store the image variable on the heap with the *Ptr_New* command, like this:

```
IDL> image = LoadData(5)
IDL> Help, image
IMAGE          BYTE      = Array [256, 256]
IDL> ptr = Ptr_New(image)
```

The variable *ptr* is the pointer reference. You can see that it is a pointer by typing this:

```
IDL> Help, ptr
PTR           POINTER   = <PtrHeapVar1>
```

And you can see what the pointer points to, by using the *Heap* keyword to the *Help* command, like this:

```
IDL> Help, /Heap
Heap Variables:
# Pointer: 1
# Object : 0

<PtrHeapVar1>    BYTE      = Array [256, 256]
```

Another way of seeing what the pointer points to is to *de-reference* the pointer. A pointer is de-referenced with the asterisk operator, like this:

```
IDL> Help, *ptr
<PtrHeapVar1>    BYTE      = Array [256, 256]
```

It is illegal to de-reference an invalid pointer. (Another name for an invalid pointer is a *null pointer*.) You can tell if you have an invalid pointer by using the *Ptr_Valid* command. A return value of 1 indicates a valid pointer, and a return value of 0 indicates an invalid or null pointer.

```
IDL> Print, Ptr_Valid(ptr)
1
```

There are two ways to create a null pointer. You can use the *Ptr_New* command with no arguments, or you can free the pointer with the *Ptr_Free* command, like this:

```
IDL> nullptr = Ptr_New()
IDL> Ptr_Free, ptr
IDL> Print, Ptr_Valid(nullptr), Ptr_Valid(ptr)
0    0
```

Notice that null pointers are invalid pointers. In other words, they cannot be dereferenced. For example, suppose you try to store another image at the pointer location now, like this:

```
IDL> image = LoadData(7)
IDL> *ptr = image
% Invalid pointer: PTR.
```

You get an error message indicating you tried to de-reference an invalid pointer.

Occasionally, however, you would like to store an undefined variable at a pointer location. (An undefined variable is a perfectly valid variable type in IDL.) The reason you would want to do so is because this kind of pointer is a valid pointer than can be de-referenced. You create a pointer to an undefined variable by using the *Allocate_Heap* keyword in the *Ptr_New* command. Type these commands:

```
IDL> newptr = Ptr_New(/Allocate_Heap)
IDL> Print, Ptr_Valid(newptr)
1
IDL> *newptr = image
IDL> Help, *newptr
<PtrHeapVar3> BYTE      = Array[360, 360]
```

Another nice thing about IDL pointers is that unlike C pointers, you don't have to worry about all kinds of memory management. IDL takes care of all of that for you. For example, if you want to change the data the variable points to, you can do so without worrying about de-allocating the current memory, allocating more memory for the new variable, storing the new variable, etc. All you have to do is something like this:

```
IDL> *newptr = [3,5,4]
IDL> Help, newptr, *newptr
NEWPTR          POINTER    = <PtrHeapVar3>
<PtrHeapVar3>   INT       = Array[3]
```

This is, of course, exactly the way memory management normally works with variables in IDL. But it is nice to know that pointers act the same way. Of course, you can get into trouble with it, too. For example, you might type this:

```
IDL> newptr = [3, 5, 4, 5, 9]
IDL> Help, newptr
NEWPTR          INT       = Array[5]
```

Whoops! Now the variable *newptr* isn't a pointer at all. It is a five-element integer array. What happened to the information that was previously stored on the heap? Well, it is still there. But you just destroyed the only reference you had to the heap data. This is what is called memory leakage, and is one of the things you want to avoid at all cost in your programming.

You can see that the memory is still allocated on the heap by looking at the heap:

```
IDL> Help, /Heap
Heap Variables:
# Pointer: 1
# Object : 0
<PtrHeapVar3>   INT       = Array[3]
```

You have a couple of options for getting rid of that leaking memory. You can use the command *Heap_GC* (for Heap Garbage Cleanup) to have IDL delete all the pointers and objects on the heap that no longer have any valid references to them. This takes some time (all current variables have to be checked), but it works. I would say this option is a last resort and is always an admission of some kind of failure on your part. I'd only use the command if I had closed the door to my office and sent my office mate down to the canteen for a couple of Danish. I would certainly never, under any circumstances, put the command into a piece of code I wrote. It would shatter the illusion I was a competent programmer.

Your other option is to call *Ptr_Valid* with the pointer index number (3, in this case) and the *Cast* keyword set. Under these circumstances, IDL will return a pointer to the leaking memory. You can then free this pointer properly. The code might look like this:

```
IDL> leakingPtr = Ptr_Valid(3, /Cast)
IDL> Ptr_Free, leakingPtr
```



Note that using *Ptr_Valid* without any arguments at all causes *Ptr_Valid* to return an array of pointers to *all* the pointers on the heap, including those that have no current reference and those that have current references. I've seen programmers try to clean up leaking memory like this:

```
IDL> leakingPtrs = Ptr_Valid()
IDL> Ptr_Free, leakingPtrs
```

This works, of course, if the program leaking memory is the only program currently running in the IDL session. But it frees valid pointers as well as ones that have lost their references. Hence, this is a dangerous practice in general.

Using Pointers in the Info Structure

So, to get back to the problem at hand, if we have information that we want to store in the *info* structure of our widget program and we either do not know how the information is defined, or the information might change its size, data type, or both during program execution, then we want to store this data as a pointer. The *info* structure, which is a local variable, should be defined like this:

```
info = { image:Ptr_New(image), $
         axisColorName:axisColorName, $
         backColorName:backcolorName, $
         binsize:binsize, $
         dataColorName:datacolorName, $
         imageColors:imagecolors, $
         max_value:max_value, $
         title:title, $
         xscale:xscale, $
         yscale:yscale, $
         extra:Ptr_New(extra), $
         drawID:drawID, $
         wid:wid $}
```

Note that the expression *Ptr_New(extra)* is the same as *Ptr_New(/Allocate_Heap)* if the *extra* variable is undefined at the time the expression is evaluated. Under these circumstances, *info.extra* is a valid pointer, pointing to an undefined variable.

Using Widget User Values to Store Program Information

The program information, to be useful, must be stored in a global memory location where it is accessible to the widget program modules that will need the information.

The variable *info* is currently a local variable that will be cleaned up and destroyed when the widget definition module code is finished executing. If you don't put the *info* structure in a global memory location, it won't exist when the event handler module needs the information.

As it happens, widgets are also created and stored in a global memory location. But even more important, each widget is created with a built-in container of sorts, called a *user value*, where you can store any kind of IDL variable. The user value is available to you for whatever purpose you choose. In this case, you can use the user value of a widget to store the *info* structure.

But which widget's user value should you use? The truth is, it doesn't really matter much. You can use the unused user value of any widget in your program for this purpose. In practice and by convention, it is the user value of the top-level base that is used for this purpose. There are advantages to this choice that will be obvious shortly.

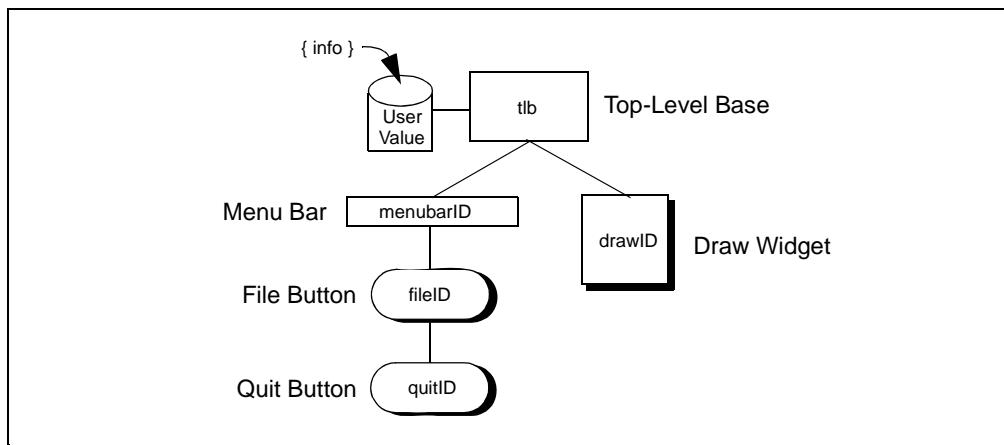


Figure 103: A diagram of the widgets created in the widget definition module of *Histo_GUI*. The *info* structure is stored in the user value of the top-level base.

To copy and store the *info* structure information in the user value of the top-level base, type this in your program file:

```
Widget_Control, tlb, Set_UValue=info, /No_Copy
```

I just said I wanted to *copy* the *info* structure into the user value of the top-level base, and yet I set a *No_Copy* keyword when I executed the command. Isn't that contradictory? What is really going on here? The secret is in practicing good memory management.

Practicing Good Memory Management

I pointed out earlier that the *info* structure in the widget definition module is a local variable. Suppose I didn't use the *No_Copy* keyword. Then the information in this local variable could be copied from this local variable into the global or protected memory location of the user value of the top-level base with this command:

```
Widget_Control, tlb, Set_UValue=info
```

The key word here is *copied*. I would actually make a duplicate copy of the *info* structure and store that in the user value of the top-level base. (The user value of the top-level base is, in effect, another variable.) And for what purpose? So I can copy the *info* structure yet again when I get into an event handler module where I need the information that is inside the *info* structure. I will have made at least three copies of the *info* structure before I am done. You see this illustrated in Figure 104.

I already mentioned that the *info* structure is where I put all the information I need to run the program. So you can imagine that in a real program there might be quite a lot of information. Making three copies of something that takes up a lot of memory is *not* good memory management, not by a long shot. But what can you do about it?

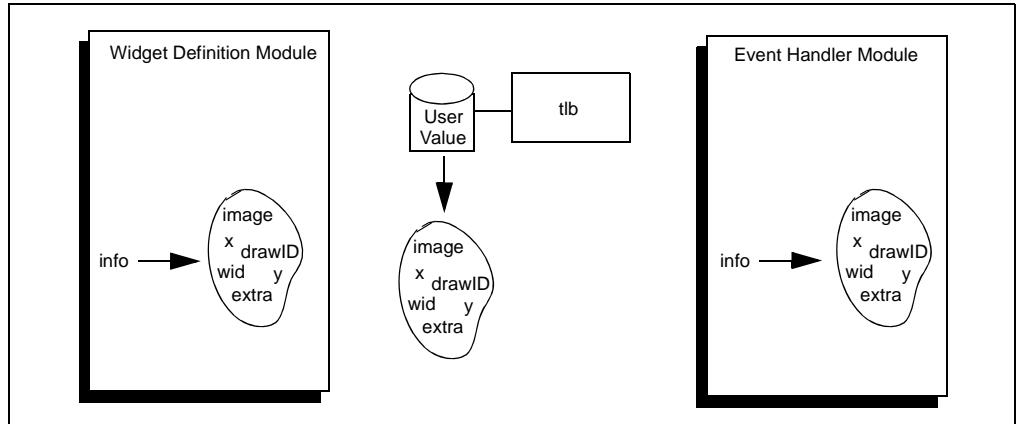


Figure 104: If you are not careful, you can make three separate copies of the same program information. This is not what you want to do. To avoid it, use the *No_Copy* keyword when you move information to and from the user value of the top-level base.

IDL has a rather strange concept of *not copying* information into variables. To understand it you have to realize a variable in IDL is a complicated C structure. One field in that structure is an actual C pointer to a memory address where data is stored. There is a rule in IDL, however, that there can only be one variable pointing to an area of machine memory at a time. (This is to keep users from crashing IDL by overwriting memory.) So, when you type these commands:

```
IDL> a = 5
IDL> b = a
```

IDL takes the information in the variable *a* and copies it into a new variable *b*. This includes making a new copy of the value 5, storing it in memory, and returning a new C pointer to that machine memory address.

However, if you wanted variable *b* to keep all the information in *a*, include its data pointer, you could do this:

```
IDL> b = Temporary(a)
```

This essentially just reuses the memory already allocated for the variable *a* in the new variable *b*. The interesting side effect of this, however, is that the variable *a* no longer has a valid data pointer (a result of the rule above allowing only one variable at a time to point to an area of machine memory). A variable without a valid data pointer is, by definition, an undefined variable.

```
IDL> Help, a
A                      UNDEFINED = <Undefined>
```

This, essentially, is what is done with the *No_Copy* keyword. By issuing this command:

```
Widget_Control, tlb, Set_UValue=info, /No_Copy
```

we define the user value of the top-level base to point to the area of machine memory where the local variable *info* was stored. But in doing so, we *undefine the variable info*. This is not a hardship, normally, since the transfer usually takes place just before

the call to *XManager*, which is the last line in the widget definition module to be executed. You are done with the *info* structure anyway. You see the effect of this operation in the illustration in Figure 105, below.

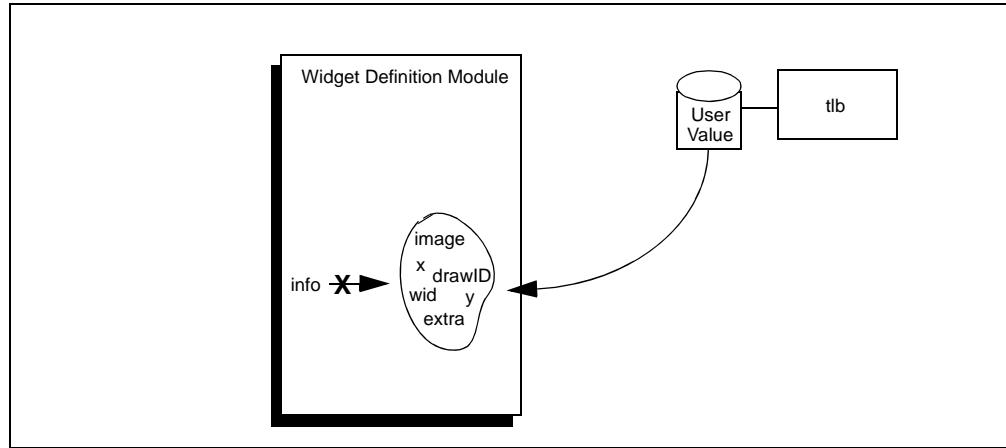


Figure 105: Transferring the *info* structure from the widget definition module to the user value of the top-level base with a *No_Copy* keyword makes the user value point to the area in memory that has already been allocated for the information. However, the local variable *info* is now undefined by this transfer process.

Similarly, you do not want to copy the information from the user value of the top-level base into the local variable *info* in an event handler module. Thus, we will use a command like this in our event handler (which we have yet to write) to transfer the information from the user value to a local *info* variable.

```
Widget_Control, event.top, Get_UValue=info, /No_Copy
```

Again, you see the effect of this operation in the illustration in Figure 106, below. The effect of this operation is to undefine the user value. Once again, this is not a problem, because it is the information in the local *info* structure that is of concern to you.

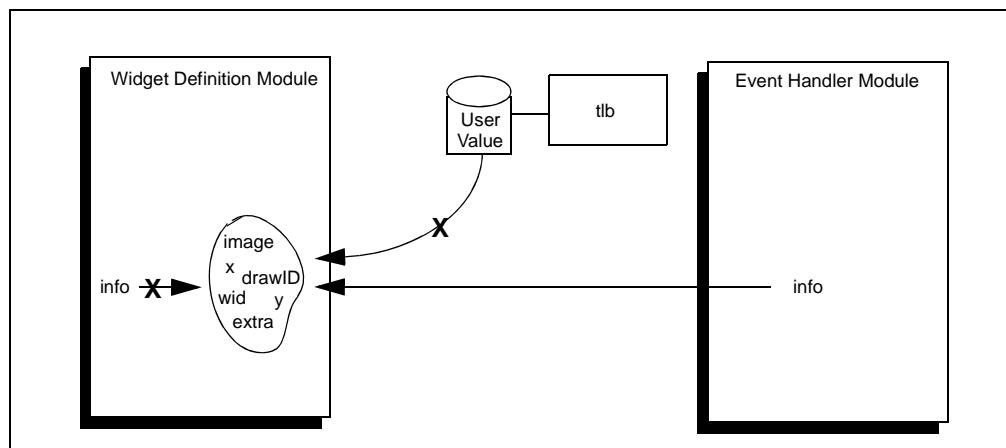


Figure 106: Transferring the information from the user value to the local *info* structure in the event handler with the *No_Copy* keyword makes the *info* variable point to the original area of memory, but has the effect of undefining the user value.

But, if this is all you do you will be in great trouble. The event handler, remember, is a one-shot event handler which processes events one at a time. The code is executed

each time there is an appropriate event to be handled. When the event is finished being processed, IDL exits the event handler module, cleaning up any local variables and the portion of memory they point to. This will include the local *info* variable and the information in its memory location.

If this happened, your event handler would only work the first time it processed an event. For subsequent events, it would look for program information in the user value of the top-level base and it would not find it there. (The user value is *undefined*.) This problem usually manifests itself in your program with the error *info must be a structure in this context*.

To avoid this problem, it is essential that before you exit the event handler module you put the *info* structure back into the user value of the top-level base with a *No_Copy* keyword. The code in your event handler (which we are yet to write) will look like this:

```
Widget_Control, event.top, Set_UValue=info, /No_Copy
```

Creating the Event Loop and Registering the Program

The final step in the widget definition module is to create the widget event loop and register the program with the window manager. If you were writing this program in C, for example, you might spend half an afternoon and have a page of code written to manage this task. In IDL it is all done with a single *XManager* command.

The first parameter to the *XManager* command is the name by which the program should be registered. This is most often the same as the widget definition module name, although it doesn't have to be. It can be any name at all. It is a good idea to spell this name in all lowercase or all uppercase letters, since the name will be case sensitive later on if you need to check it for some reason.

The second parameter to the *XManager* command is the widget identifier of the top-level base widget. By knowing the identifier of the top-level base, *XManager* will know all the other widgets that belong in that hierarchy.

Normally, widget programs block the IDL command line and prevent the user from typing IDL commands while the program is running. This is the way all widget programs behaved before IDL 5.0. There are many advantages to having widget programs block the command line, but users begged Research Systems to allow them to run their widget programs *and* have access to the IDL command line. Finally, Research System capitulated. It is both good news and bad news.

It is nice to have access to the IDL command line, but it means all kinds of bad things can happen to widget programs. The user could, for example, totally ruin your beautiful color scheme by loading some other color table from the command line. So one of the things we have to learn to do is protect our widget programs from command line sabotage. You will learn more about this in a moment. But for now we will allow command line access by setting the *No_Block* keyword.

I mentioned that you should never use either the *Event_Pro* or *Event_Func* keyword to assign the event handler for the top-level base. Actually, this warning does not apply so much to top-level bases as it does to top-level bases that are being directly managed by *XManager*. (Ninety-five percent of top-level bases *are* being directly managed by *XManager*.) The proper way to assign an event handler module, and it must be a procedure, to the top-level base is with the *Event_Handler* keyword, as shown here.



Note that in the absence of a *Event_Handler* keyword, the event handler will be assigned automatically to this top-level base. The event handler name will be constructed from the program's register name (*histo_gui*, in this case), with an *_event* appended to it. In other words, without the *Event_Handler* keyword, the name of the

event handler module associated with this top-level base would be name *Histo_GUI_Event*.

Finally, the presence of pointers in this program presents a perfect opportunity to have memory leaking like a sieve. To prevent that we have to free our pointers properly when we exit the program. We are going to do this with a clean-up procedure. We name that procedure by using the *Cleanup* keyword on the *XManager* command.

To register the program and end the widget definition module, type these two commands:

```
XManager, 'histo_gui', tlb, /No_Block, $  
    Event_Handler='Histo_GUI_TLB_Events', $  
    Cleanup='Histo_GUI_Cleanup'  
  
END
```

Running the Program

At this stage you can compile and run your *Histo_GUI* program, but be careful not to generate any events.

```
IDL> .Compile histo_gui  
IDL> Histo_GUI
```

If the program doesn't compile or run, there are probably errors in the code. Remove the widget program from the display with your mouse, type *RetAll* (*Return All* the way back to the main program level), fix the errors, and try again. Remember this sequence of operations, because you should do this *every* time you crash a widget program.

What would happen if you did generate an event? Try it. Why does this happen?

Only two widgets can generate events in this program: the top-level base when it is resized by the user, and the *Quit* button when it is selected by the user. Draw widgets can generate events, but they must be explicitly set up to do so and that has not been done here. The *File* button cannot generate events because it is a menu button (i.e., it has the *Menu* keyword set for it.)

If you try to generate an event in the *Histo_GUI* program an error occurs. This is because you haven't written the event handler module for this program. You will do that in just a moment.

Close your *Histo_GUI* program with your mouse. Look at your heap area.

```
IDL> Help, /Heap  
Heap Variables:  
    # Pointer: 1  
    # Object : 0  
  
<PtrHeapVar5>   BYTE      = Array[360, 360]
```

Oh, oh. Memory leaks already. Close your office door and get that *Heap_GC* command ready!

```
IDL> Heap_GC
```

Now, keep reading.

Writing the Event Handler Modules

The purpose of the event handler modules is to handle the events the program generates. Recall that it is the window manager that is responsible for knowing about

and managing events. When an event occurs, the window manager packages information about the event up and sends it to IDL. IDL takes the event information and re-packages into an event structure, which is then passed directly to the proper event handler module as its one and only positional parameter or argument.

The event structure has different fields in it, depending upon which widget caused the event. (See *Appendix A: Widget Event Structures* for more information about widget event structures.) All widget event structures, however, have three fields in common: *ID*, *Top*, and *Handler*. It is important to know what these three fields mean.

Common Fields in Event Structures

The *ID* field in an event structure is always the identifier of the widget that caused the event. For example, if the user selects the *Quit* button in the program you are writing, then the button event structure will have its *ID* field set to the identifier of the *Quit* button (i.e., the value you call *quitID* in the code above).

The widget that caused the event always exists in a hierarchy of other widgets. The *Top* field identifies the widget located at the top of that hierarchy. In other words, it is the identifier of the top-level base widget of whatever widget hierarchy the widget identified by the *ID* field is associated with. It would be the widget identified by *tlb* in the example above.

The *Handler* field is slightly more complicated. You know that every widget program must have at least one event handler module. But real widget programs often have several event handler modules. In fact, you can associate an event handler with every widget you create, although programs are not usually written this way. Separate event handlers are often called for if the “actions” required by separate events are sufficiently different from one another in the algorithms they use to accomplish their tasks.

Each event handler created in a widget program must be associated with a particular widget. The *Handler* is the widget identifier of the widget associated with that specific event handler that will be handling the events of the widget that caused the event. Got that?

A picture makes it *much* easier to understand. Consider the illustration in Figure 107 in which the widgets A-G are defined. There is an event handler associated with the top-level base, widget A. Suppose an event occurs at widget G. IDL looks to see if there is an event handler associated with widget G. In this case there is not. So the event is said to “bubble up” the widget hierarchy. IDL looks at widget F. Is there an event handler associated with this widget? No. The event bubbles up to widget C, etc. Eventually, the event finds an event handler associated with widget A.

The event structure, then, is filled out like this. The *ID* field identifies the widget that caused the event: widget G. The *Top* field identifies the widget at the top of the widget hierarchy to which widget G belongs: widget A. The *Handler* field identifies the widget associated with the event handler handling the events that come from widget G: also widget A.

But consider a slightly more complicated widget program as illustrated in Figure 108. Here there are two event handlers: one associated with widget A and one with widget F. If an event occurs at widget G, the event bubbles up to the event handler associated with widget F. The *Handler* field of the event structure now identifies widget F instead of widget A.

What happens when the event from widget G goes into the event handler associated with widget F depends upon how the event handler at widget F is written.

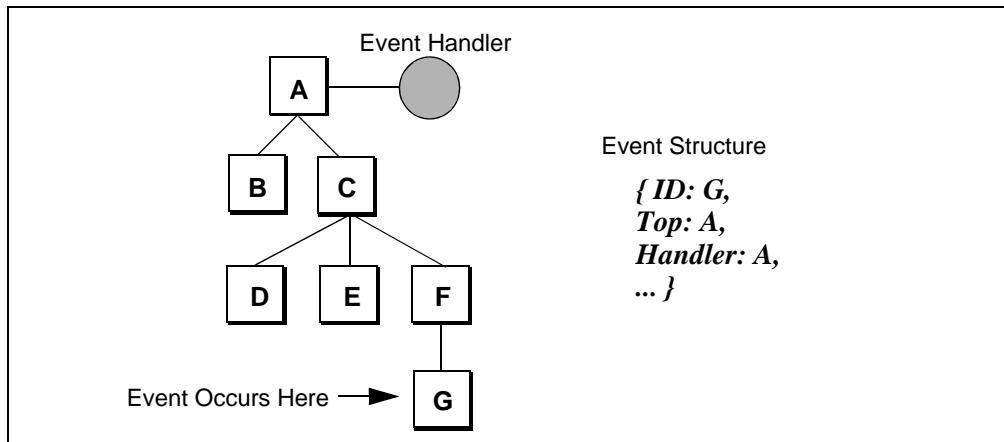


Figure 107: A simple widget program with widgets A through G defined. An event handler (filled circle) is associated with the top-level base. If an event occurs at widget G, the event bubbles up to the event handler associated with widget A.

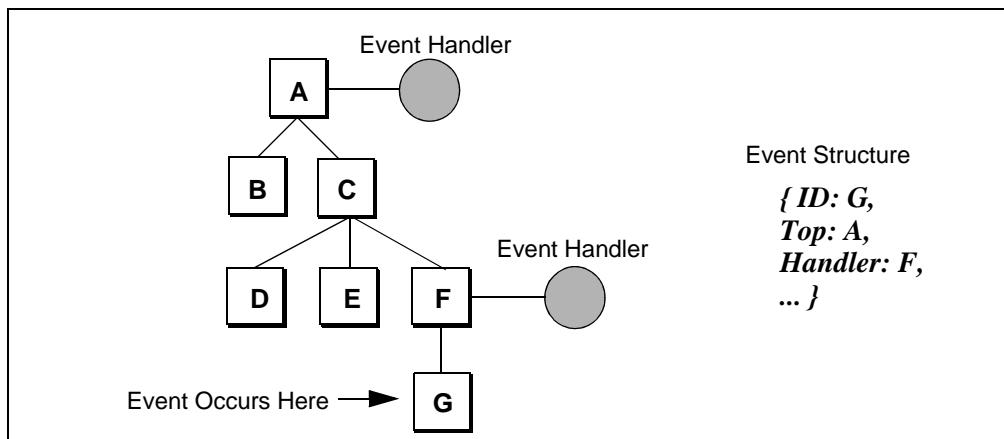


Figure 108: A slightly more complicated widget program with two event handler modules. Events originating at widget G are handled by the event handler associated with widget F.

Event Handler Functions

Event handler modules can be either procedures or functions. If the event handler module associated with widget F is a procedure, the event is said to be *swallowed* by the event handler. That is to say, the event is processed and that is the end of it. The program waits for the next event to occur. But suppose, for example, that the event handler is a function rather than a procedure. Functions, you recall, return a value. If the return value of an event handler function is a structure, and that structure has the fields *ID*, *Top*, and *Handler* defined as long integers, then IDL will treat the return value of that event handler function as an event that will bubble up to the next event handler in the hierarchy. If the return value does not meet the criteria for an event structure, the event is said to be swallowed by the event handler function.

This makes it possible to selectively return events, or to process events and create your own events. Suppose, for example, you wanted a draw widget to act as a button. You could take the draw widget event into the event handler and repackage it as a button event and pass that button event structure to the next event handler in the hierarchy.

The tail-end of such a pseudo-button event handler function might look like this:

```

pseudoButtonEvent = { PSEUDO_BUTTON_EVENT, $
                      ID:event.handler, $
                      Top:event.top, $
                      Handler:0L, $
                      Select:1L }
RETURN, pseudoButtonEvent
END

```

Notice the *Handler* field is filled out with the long integer 0, an invalid widget identifier. IDL will take this return value, see that the *ID*, *Top*, and *Handler* fields are defined properly as long integers, and will then fill in the proper value for the *Handler* field. You do not have to concern yourself with identifying the proper handler initially. It is important, however, to get proper widget identifiers into the *ID* and *Top* fields of such a return value. The values shown in this example are typical.

Associating Event Handlers with Widgets

But how can event handler modules be associated with particular widgets? The simplest way is to use the keyword *Event_Pro* (if the event handler module is a procedure) or the keyword *Event_Func* (if the event handler module is a function) to assign the proper module when the widget is created.

For example, in the code above you created a separate event handler procedure, named *Histo_GUI_Quit*, for the *Quit* button by using the *Event_Pro* keyword in the button creation routine.



It is a good idea to give your event handler modules unique names. I like to preface the event handler names with the widget definition module name. (This is also sometimes known as the *command* name.) Then there is never confusion about which IDL module should be called. The name is not case sensitive.

The *Event_Pro* and *Event_Func* keywords can be used to associate an event handler module with any widget except a widget that is being managed directly by *XManager* (i.e., a top-level base whose identifier is a parameter to the *XManager* command).

Top-level base widgets managed by *XManager* must have their event handlers assigned to the widget with the *Event_Handler* keyword on the *XManager* command. The event handler associated with the top-level base must *always* be a procedure. It can not be a function.

If the *Event_Handler* keyword is not used on the *XManager* call then the default name of the event procedure associated with the top-level base is created by using the program's register name with an *_Event* attached to it. For example, the default event handler name for the *Histo_Gui* program is *Histo_GUI_Event*, since the register name (the first parameter in the *XManager* command) is 'histo_gui'. Notice that the event handler name is not case sensitive.

We often give the event handler associated with the top-level base a more descriptive name that reflects its function in the program. Since the top-level base event handler is the *Histo_Gui* program will eventually be used to both resize the graphics window and to protect the programs colors, we named it *Histo_GUI_TLB_Events*. You see an illustration of the widget program now in Figure 109, with the event handlers now associated with the proper widgets.

Writing the Quit Button Event Handler

The *Quit* button event handler is simple to write. It is a procedure in this case. Its only purpose is to destroy all of the widgets and remove them from the display. If you destroy the top-level base, you will automatically destroy all of the widgets in that hierarchy. (This is just the opposite of realizing the top-level base.)

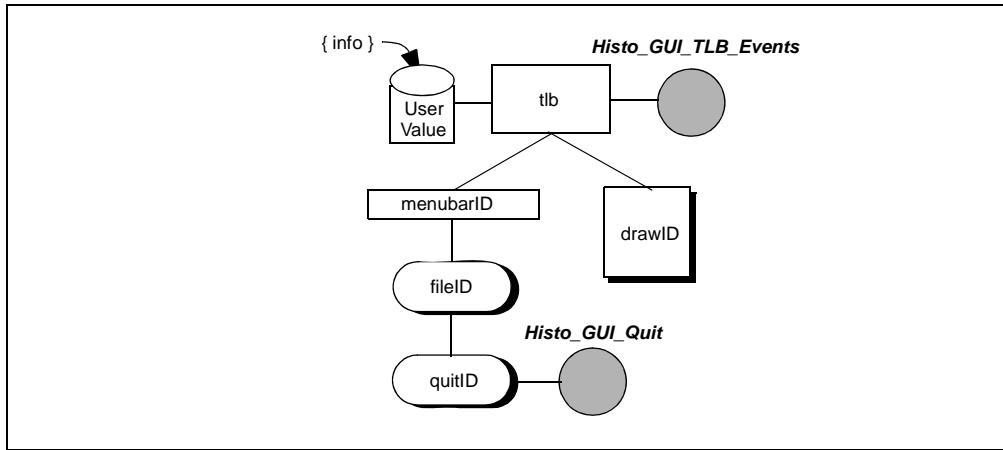


Figure 109: The event handler modules are now associated with specific widgets in this diagram of the *Histo_GUI* program.

The only trick is that you do not have the identifier of the top-level base, *tlb*, available to you in this event handler. The variable *tlb* is a local variable in the widget definition module. That variable was destroyed as soon as the code for the widget definition module was executed. What you have in this module is the *equivalent* of *tlb*. You have the *Top* field of the event structure.

The entire event handler, defined with one and only one positional parameter (which is the event structure that is passed to it from IDL), looks like this. Please type this code *before* the widget definition module code in the file *histo_gui.pro*.

```

PRO Histo_GUI_Quit, event
Widget_Control, event.top, /Destroy
END

```

Note that the event structure does not have to be named *event*. You can give it any name you like. I always call it *event* so that I know what I am working with when I see it in my widget program code.

Writing the Resizeable Graphics Window Event Handler

The event handler to resize the graphics window is slightly more complicated than the *Quit* button event handler. Think what needs to happen if the top-level base is resized.

1. You need to know the final size of the newly-sized graphics window. This information will come from the *X* and *Y* fields in the base widget event structure. (See “Base Widget Event Structure” on page 389 for a description of a base widget event structure.)
2. You need to make the draw widget (i.e., the graphics window) the proper size. This can be done with the *Widget_Control* command if you know the draw widget’s identifier. In this case, the draw widget’s identifier is *drawID*.
3. You must make the draw widget the current graphics window. This can be done with the *WSet* command if you know the draw widget’s graphics window index number. In this case, the draw widget’s graphics window index number is *wid*.
4. You must redraw the graphic display. This can be done with the *HistoImage* command if you know the data and keyword parameters to pass into the command.

The information for requirement 1 will come from the event structure itself. Recall from Appendix A or from the on-line help for *Widget_Base* that the base widget event structure for a top-level base resize event is defined like this:

```
event = { WIDGET_BASE, ID:0L, Top:0L, Handler:0L, X:0, Y:0 }
```

The *X* and *Y* fields are the final size of the base in pixels. In most versions of IDL (certainly all recent versions), the size of the top-level base reported in the event structure does not include the part of the top-level base taken up by either the menu bar or by other window decorations. In other words, the final top-level base size is the size we want to make the draw widget.

The information for requirements 2-4 will come from the information stored in the *info* structure. Remember that the *info* structure was stored in a global memory location in IDL. All you need to know to access it and copy its contents into a local variable in this event handler is its location. It is in the user value of the top-level base, which will be identified by the *Top* field of the event structure.

Open the *histo_gui.pro* file, and type the following two lines of code at the *beginning* of the file. The first line is the event handler procedure definition line. The second line of code copies the *info* structure from its storage location in the user value of the top-level base into a local variable named *info*. Type:

```
PRO Histo_GUI_TLB_Events, event
Widget_Control, event.top, Get_UValue=info, /No_Copy
```

Note that we used the *No_Copy* keyword to take the *info* structure out of its storage location in the user value of the top-level base and copy it to the local variable *info*. (We could have named this local variable anything we like. But calling it *info* makes it less confusing, as long as we understand what we are doing. Just be aware that this variable is a *local* variable to the *Histo_GUI_TLB_Events* procedure.)

Next, resize the draw widget with the *Draw_XSize* and *Draw_YSize* keywords so that it is the same size as the top-level base. Note that the top-level base size that is returned in the event structure fields does not include either the menu bar space or the window title or frame area. You will need the draw widget's identifier, which is stored in the local *info* structure, as well as the correct sizes from the *event* structure. Type:

```
Widget_Control, info.drawID, Draw_XSize=event.x, $
Draw_YSize=event.y
```

Next, make the draw widget the current graphics window so you can display the graphics in this window.

```
WSet, info.wid
```



The importance of this step cannot be overemphasized. In widget programs, you *must* make sure you know which window you are directing graphics into or you will end up drawing graphics in some other widget program's window. *Always* issue a *WSet* command before you draw *any* graphics in a draw widget window!

Finally, you are ready to re-display the graphics display. In this case, it is important to set the *NoLoadCT* keyword to the *HistoImage* command, or you will be unable to control the image colors from the *Histo_GUI* program. Type:

```
HistoImage, *info.image, $
AxisColorName=info.axisColorName, $
BackColorName=info.backcolorName, $
Binsize=info.binsize, $
DataColorName=info.datacolorName, $
_Extra=*info.extra, $
Max_Value=info.max_value, $
NoLoadCT=1, $
XScale=info.xscale, $
YScale=info.yscale
```

Finally, put the *info* structure back into the storage location in the user value of the top-level base. If you fail to do this, your program will work for the first event and fail for the second with the error *INFO must be a structure in this case*.

```
Widget_Control, event.top, Set_UValue=info, /No_Copy  
END
```



If you run a widget program and you get the error message *INFO must be a structure in this case*, then 99 times out of 100 you have made one of two errors: (1) you have checked the *info* structure out of the user value of the top-level base with a *No_Copy* keyword, but you have forgotten to return it to the user value before you exited the event handler, or (2) you have tried to use the *info* structure after you have checked it into the user value with a *No_Copy* keyword, in which case it is currently an undefined variable. You can identify the first error is your program processes one event, but not two in a row. Look for the error in the previous event handler that *worked*, not the one that just failed!

Writing the Cleanup Procedure

The last step is to write the *Histo_GUI_Cleanup* procedure. The *Histo_GUI_Cleanup* procedure is similar to an event handler, but it is not sent an event structure as the one and only positional parameter as is an event handler module. Rather, this procedure is called with the identifier of the widget that is associated with the clean-up procedure when that widget dies. In other words, when the top-level base is destroyed, this clean-up procedure will be called with the identifier of the top-level base as a parameter. (The correct term for this module is a *callback* procedure, rather than an event handler.)

Since the top-level base widget has already disappeared from the display, there is not much we can do with it. In fact, the only thing we can do with this top-level base is inquire about its user value. But, fortunately, this is what we need to know to clean our program up properly.

Add the following code to get the *info* structure out of the user value of the top-level base somewhere in front of your widget definition module in the program file *histo_gui.pro*.

```
PRO Histo_GUI_Cleanup, tlb  
Widget_Control, tlb, Get_UValue=info, /No_Copy
```

The clean-up procedure will be called whenever the top-level base is destroyed. This could be when the *Quit* button is selected (unusual) or it could be when the user kills the top-level base with the mouse (usual). It will even be called if your program crashes and the user kills the top-level base with the mouse. Under these circumstances, it might be that the *info* structure is not in the user value of the top-level base. You always want to check for that eventually, or your clean-up procedure will cause another error when you are trying to clean up from a previous error. Your code might look like this:

```
IF N_Elements(info) EQ 0 THEN RETURN  
Ptr_Free, info.image  
Ptr_Free, info.extra  
END
```

Note that if the *info* structure is not defined, you are going to return immediately out of the procedure. It will be the user's responsibility to clean up leaking memory in such a case. (Don't worry, this won't happen when you get your programs working finally, because we are going to anticipate any error that might have occurred and make sure the *info* structure is where it is suppose to be.)

The proper sequence of commands to recover from this error might be something like this:

```
IDL> RetAll
IDL> Heap_GC
```

If the *info* structure is defined (it should be) then you will clean up the appropriate pointers yourself. The clean-up routine is where you clean up anything that may take up memory: pointers, objects, pixmaps if you have used them, etc.

Note that you do not have to put the *info* structure back into the user value. There is really no need here, since the top-level base has already been destroyed and the *info* structure itself will be cleaned up as soon as the program exits.

Running the Program

This program is ready to be compiled and run. Test it to see if the graphics window resizes properly and that the *Quit* button works. (If you did not enter the program above into a file named *histo_gui.pro*, you can use the file *histo_gui.1.pro*, which you downloaded to use with this book. This file must be explicitly compiled to be used. See “The Advantage of Mistakes” on page 261 for more information.)

```
IDL> .Compile Histo_GUI
IDL> Histo_GUI
```



If you are running IDL on a Windows 95 or Windows NT operating system, you may notice that the window flickers and resizes slowly. This is because the “Show window contents while dragging” option is turned on for your display. You should disable this option. Open the *Display* control panel (you can right click on your desktop, if you like) and select the *Plus* tab. In the *Visual Settings* section of the control panel, de-select the “Show window contents while dragging” option.

Recovering From Program Errors

If the program either doesn’t compile or doesn’t run correctly, it is likely because you introduced program errors while you were typing it. There is a correct sequence of steps to take to recover from errors in widget programs. Programmers who don’t observe this sequence sooner or later find themselves frustrated beyond belief at how obstinate IDL can be. I am warning you at your peril. The correct sequence of steps to recover from an error in a widget program are these.

1. Remove the widget program from the display with the mouse. I can’t emphasize enough how important this first step is. If you ever find yourself swearing at your program, threatening the fine folks at Research Systems, or kicking your dog, check to see how many “extra” programs you still have on your display. If the answer is “more than zero”, that is your problem, almost certainly. IDL can become very, very confused if you repeatedly run the same widget program over and over without cleaning up the ones that break. Remove the crashed programs from the display with your mouse.
2. Get used to the following command and use it religiously: *RetAll*. It means “Return All the way back to the main IDL level.”

```
IDL> RetAll
```

This single command is the most used command in the IDL language. If it’s not, you are writing programs that are way too easy for you. Try something harder.

3. Fix the error in your program, re-compile, and try running it again.

If you follow these three steps you can get your widget programs running again 99.9% of the time. You do *not* have to exit IDL and start all over, as too many beginning widget programmers are prone to do.

Very occasionally (and especially if you are running older versions of IDL), you can cause a crash that widget programs just can't seem to recover from. On those occasions (and they haven't happened to me in over two years now), you might want to throw in a single *XManager* command right after the *RetAll*. This will really wake the widgets up. But don't do it too often or they will be so hyped up they will be hard to handle.

```
IDL> XManager
```

Do these three steps appear to be the same three steps you should use to recover from *any* programming error? Uh, well ... yes. But you would be surprised at how often people who are writing widget programs forget them. Following them will guarantee success. Well ... more or less. Read on.

Adding Color Protection

The *Histo_GUI* program is so simple that you might imagine it is perfect just the way it is. But, alas, life is not so accommodating. It's not bad. It resizes nicely. You can quit it properly. It cleans up after itself. But remember we wrote it as a non-blocking widget program by setting the *No_Block* keyword on the *XManager* command at the very bottom of the widget definition module.

Giving the user access to the IDL command line while a widget program is running can be the kiss of death to many widget program. Ours is not so bad, but remember we are writing this program so that image colors can be controlled externally. (We haven't actually tried to control them yet, but that is our goal eventually.)

To see what the problem might be, try this. Start your program up.

```
IDL> Histo_GUI
```

Notice that the program starts with color table 4, the default color table for the *HistoImage* program. But then load a different color table from the IDL command line.

```
IDL> LoadCT, 3
```

If you are on an 8-bit display, the colors (both image colors and drawing colors) in the display window changed immediately. On a 24-bit display nothing bad has happened yet. But now resize the graphics window.

Whoops! The image is now using the red-temperature scale colors. This is not what we want. We want to keep using the original color table until the program decides to change it. In other words, we want to protect our program's colors.

There are various ways to do this, but they all involve the program keeping track of its own color vectors and restoring them at the proper time. I am going to demonstrate a method of color protection that involves keyboard focus events. In other words, when this program gets the keyboard focus (the user toggles the program display window forward on their display, they click in the display window, etc.) then the program's color table will be reloaded and the program will update itself.

I can't prevent the program from momentarily having improper colors when I give the user access to the command line. I can only make sure that when the user *interacts* with the program in some way, that the proper colors are loaded.

Saving the Color Vectors

The first step in color protection involves saving the proper color vectors. When do we have the proper colors in this program? Right, just after we call the *HistoImage* program the first time in the widget definition module to display the graphics. We must collect those RGB color vectors at that time and save them in our *info* structure.

Find these lines in your widget definition program code:

```
info = { image:Ptr_New(image), $  
        axisColorName:axisColorName, $  
        backColorName:backcolorName, $  
        binsize:binsize, $  
        dataColorName:datacolorName, $  
        imageColors:imagecolors, $  
        max_value:max_value, $  
        xscale:xscale, $  
        yscale:yscale, $  
        extra:Ptr_New(extra), $  
        drawID:drawID, $  
        wid:wid $  
    }
```

Add the text written in bold below to recover the RGB vectors and store them in the *info* structure. The *Get* keyword to the *TVLCT* command will cause the current color table vectors to be loaded into the variables, *r*, *g*, and *b*.

```
TVLCT, r, g, b, /Get  
info = { image:Ptr_New(image), $  
        axisColorName:axisColorName, $  
        backColorName:backcolorName, $  
        binsize:binsize, $  
        dataColorName:datacolorName, $  
        imageColors:imagecolors, $  
        max_value:max_value, $  
        title:title, $  
        xscale:xscale, $  
        yscale:yscale, $  
        extra:Ptr_New(extra), $  
r:r, $  
g:g, $  
b:b, $  
        drawID:drawID, $  
        wid:wid $  
    }
```

Setting Up Keyboard Focus Events

The next step is to set up keyboard focus events. These events can be set for the top-level base widget, but they are not turned on by default. Like the resize events, we must explicitly turn them on. We do this by setting the *KBRD_Focus_Events* keyword for the top-level base.

We can set this keyword in the *Widget_Base* creation routine at the time we create the top-level base, or we can set it later with *Widget_Control*. In this case, we will set it at the same time we put the *info* structure into the user value of the top-level base. By doing it after the top-level base is realized, we prevent our graphics from being displayed in the window twice.

Just after the line creating the *info* structure in your widget definition module (the line you just modified in the code above), find this line of code:

```
Widget_Control, tlb, Set_UValue=info, /No_Copy
```

Change this to also turn on keyboard focus events for the top-level base:

```
Widget_Control, tlb, Set_UValue=info, /No_Copy, $  
/KBRD_Focus_Events
```

Now that keyboard focus events are on, we have to modify the event handler for the top-level base to respond to them properly.

Modifying the Histo_GUI_TLB_Events Event Handler

At the moment, the *Histo_GUI_TLB_Events* event handler is set up to respond to top-level base resize events. But now we are going to have to modify it to respond to both resize events and keyboard focus events. How should we tell the two events apart?

Perhaps the easiest way is by their name. Recall that a resize event structure is defined like this:

```
event = {WIDGET_BASE, ID:0L, Top:0L, Handler:0L, X:0, Y:0}
```

A keyboard focus event structure is defined like this:

```
event = {WIDGET_KBRD_FOCUS, ID:0L, Top:0L, Handler:0L, $  
Enter:0}
```

The *Enter* field is set to 0 if the widget is losing focus and to 1 if the widget is gaining focus. We will be concerned only if the widget is gaining focus.

We can use the IDL *Tag_Names* command to examine the event structure and return the name of the structure to us. If the name is WIDGET_BASE we will know this is a resize event. If the name is WIDGET_KBRD_FOCUS we know this is a keyboard focus event.

The code for the *Histo_GUI_TLB_Events* event handler re-written to respond to just resize events will look like this. The new code is written in bold.

```
PRO Histo_GUI_TLB_Events, event  
thisEvent = Tag_Names(event, /Structure_Name)  
IF thisEvent EQ 'WIDGET_BASE' THEN BEGIN  
    Widget_Control, event.top, Get_UValue=info, /No_Copy  
    Widget_Control, info.drawID, Draw_XSize=event.x, $  
        Draw_YSize=event.y  
    WSet, info.wid  
    HistoImage, *info.image, $  
        AxisColorName=info.axisColorName, $  
        BackColorName=info.backcolorName, $  
        Binsize=info.binsize, $  
        DataColorName=info.datacolorName, $  
        _Extra=*info.extra, $  
        Max_Value=info.max_value, $  
        NoLoadCT=1, $  
        XScale=info.xscale, $  
        YScale=info.yscale  
    Widget_Control, event.top, Set_UValue=info, /No_Copy  
ENDIF  
END
```



Note that *Tag_Names* always returns the name of the structure in uppercase characters. This is important if you are going to be doing some kind of string comparison, such as in the *IF* statement shown or in a *CASE* statement.

We are now ready to add the code to respond to the keyboard focus events. First of all, we only want to respond if this widget is gaining focus. We could try to do something if the widget is losing focus, but such attempts are charged with uncertainty. It is better to have the philosophy that each widget will look out for itself. This makes it possible for many different widget programs to work together successfully.

Then, we clearly want to restore or reload the program's color vectors. Finally, if we are running on a 24-bit display, we want to re-display the graphic. If we don't do this, there is no guarantee the display will be shown in the correct colors. Insert this code just before the final *END* statement in the *Histo_GUI_TLB_Events* event handler:

```

IF thisEvent EQ 'WIDGET_KBRD_FOCUS' THEN BEGIN
    IF event.enter EQ 0 THEN RETURN
    Widget_Control, event.top, Get_UValue=info, /No_Copy
    TVLCT, info.r, info.g, info.b
    Device, Get_Visual_Depth=theDepth
    IF theDepth GT 8 THEN BEGIN
        WSet, info.wid
        HistoImage, *info.image, $
            AxisColorName=info.axisColorName, $
            BackColorName=info.backcolorName, $
            Binsize=info.binsize, $
            DataColorName=info.datacolorName, $
            _Extra=*info.extra, $
            Max_Value=info.max_value, $
            NoLoadCT=1, $
            XScale=info.xscale, $
            YScale=info.yscale
    ENDIF
    Widget_Control, event.top, Set_UValue=info, /No_Copy
ENDIF
END

```

Test this program to see if the colors are truly protected. (If you did not enter the program above into a file named *histo_gui.pro*, you can use the file *histo_gui.2.pro*, which you downloaded to use with this book. This file must be explicitly compiled to be used. See “The Advantage of Mistakes” on page 261 for more information.) Run several versions of *Histo_GUI* at the same time with different color tables. Are the correct color tables loaded when the focus is in the right window? What happens when the user loads a color table from the IDL command line?

```

IDL> Histo_GUI, Colortable=3
IDL> Histo_GUI, Colortable=1
IDL> LoadCT, 0

```

Buffering the Graphic Display for Smoother Output

Although the graphical display portion of this program works fine, the *HistoImage* program is fairly complicated to render. As a result, when the window is resized, there is a bit of flicker as the graphics are rendered. Although not particularly bothersome on computers with fast graphics display cards, it is possible to improve rendering performance on all computers by using a technique called *double buffering*. In this technique, the graphic is rendered in an off-screen buffer, and then transferred to the display device all at once.

Double buffering in IDL is easily accomplished with a pixmap window and the *Device Copy* technique of transferring data from the pixmap to the display window. (See “The Device Copy Method of Erasing Annotation” on page 116 for additional information about this technique.) Setting up a double buffering capacity in this program simply requires that we create a pixmap and render the graphics there, before transferring the entire pixmap contents to the display window.

Find these lines, which displays the graphics in the display window, in your widget definition module:

```
Widget_Control, drawID, Get_Value=wid
WSet, wid

HistoImage, image, $
AxisColorName=axisColorName, $
BackColorName=backcolorName, $
Binsize=binsize, $
ColorTable=colortable, $
DataColorName=datacolorName, $
_Extra=extra, $
ImageColors=imagecolors, $
Max_Value=max_value, $
XScale=xscale, $
YScale=yscale
```

Rather than drawing directly into the display window, we will create a pixmap window and draw the graphics into that. Modify the code above to look like this:

```
Widget_Control, drawID, Get_Value=wid
Window, XSize=400, YSize=400, /Pixmap, /Free
pixID = !D.Window

HistoImage, image, $
AxisColorName=axisColorName, $
BackColorName=backcolorName, $
Binsize=binsize, $
ColorTable=colortable, $
DataColorName=datacolorName, $
_Extra=extra, $
ImageColors=imagecolors, $
Max_Value=max_value, $
XScale=xscale, $
YScale=yscale .
```

Then, we will copy the information in the pixmap window to the display, with this code, added directly after the *HistoImage* command above:

```
WSet, wid
Device, Copy=[0, 0, !D.X_Size, !D.Y_Size, 0, 0, pixID]
```

We will want to perform this double buffering trick every time we render graphics to the display window. Thus, we will need to keep track of the pixmap window identifier and store it in the *info* structure. Modify your *info* structure like this:

```
info = { image:Ptr_New(image), $
axisColorName:axisColorName, $
backColorName:backcolorName, $
binsize:binsize, $
dataColorName:datacolorName, $
imageColors:imagecolors, $
```

```

    max_value:max_value, $
    title:title, $
    xscale:xscale, $
    yscale:yscale, $
    extra:Ptr_New(extra), $
    r:r, $
    g:g, $
    b:b, $
    drawID:drawID, $
    wid:wid, $
    pixID:pixID $
}

```

There are two other places where we are displaying graphics in this program, both in the *Histo_GUI_TLB_Events* event handler. First, consider the case in which the top-level base is resized. In this case, you resize the draw widget. Unfortunately, it is impossible to resize a pixmap window. The only alternative is to delete the last pixmap window and create a new one of the proper size. Find this code in the *Histo_GUI_TLB_Events* event handler:

```

IF thisEvent EQ 'WIDGET_BASE' THEN BEGIN
    Widget_Control, event.top, Get_UValue=info, /No_Copy
    Widget_Control, info.drawID, Draw_XSize=event.x, $
        Draw_YSize=event.y
    WSet, info.wid
    HistoImage, *info.image, $
        AxisColorName=info.axisColorName, $
        BackColorName=info.backcolorName, $
        Binsize=info.binsize, $
        DataColorName=info.datacolorName, $
        _Extra=*info.extra, $
        Max_Value=info.max_value, $
        NoLoadCT=1, $
        XScale=info.xscale, $
        YScale=info.yscale
    Widget_Control, event.top, Set_UValue=info, /No_Copy
ENDIF

```

Change the code to relocate the *WSet* command and add the following lines (shown in bold) below:

```

IF thisEvent EQ 'WIDGET_BASE' THEN BEGIN
    Widget_Control, event.top, Get_UValue=info, /No_Copy
    Widget_Control, info.drawID, Draw_XSize=event.x, $
        Draw_YSize=event.y
    WDelete, info.pixID
    Window, XSize=event.x, YSize=event.y, /Pixmap, /Free
    info.pixID = !D.Window
    HistoImage, *info.image, $
        AxisColorName=info.axisColorName, $
        BackColorName=info.backcolorName, $
        Binsize=info.binsize, $
        DataColorName=info.datacolorName, $
        _Extra=*info.extra, $
        Max_Value=info.max_value, $
        NoLoadCT=1, $
        XScale=info.xscale, $

```

```

    YScale=info.yscale
WSet, info.wid
Device, Copy=[0, 0, !D.X_Size, !D.Y_Size, 0, 0, info.pixID]
Widget_Control, event.top, Set_UValue=info, /No_Copy
ENDIF

```

The other change occurs in the code that handles keyboard focus events. Find this code in the *Histo_GUI_TLB_Events* event handler:

```

IF theDepth GT 8 THEN BEGIN
    HistoImage, *info.image, $
    AxisColorName=info.axisColorName, $
    BackColorName=info.backcolorName, $
    Binsize=info.binsize, $
    DataColorName=info.datacolorName, $
    _Extra=*info.extra, $
    Max_Value=info.max_value, $
    NoLoadCT=1, $
    XScale=info.xscale, $
    YScale=info.yscale
ENDIF

```

Relocate the *WSet* command and add the following lines (shown in bold) below:

```

IF theDepth GT 8 THEN BEGIN
    WSet, info.pixID
    HistoImage, *info.image, $
    AxisColorName=info.axisColorName, $
    BackColorName=info.backcolorName, $
    Binsize=info.binsize, $
    DataColorName=info.datacolorName, $
    _Extra=*info.extra, $
    Max_Value=info.max_value, $
    NoLoadCT=1, $
    XScale=info.xscale, $
    YScale=info.yscale

WSet, info.wid
Device, Copy=[0, 0, !D.X_Size, !D.Y_Size, 0, 0, info.pixID]
ENDIF

```

The final change you have to make is to be sure to delete the program pixmap in the *Histo_GUI_Cleanup* routine. Remember, the purpose of the clean-up routine is take care of anything that might cause memory leakage. Un-deleted pixmaps will certainly cause memory to be used inefficiently.

Find this code in the *Histo_GUI_Cleanup* program module:

```

IF N_Elements(info) EQ 0 THEN RETURN
Ptr_Free, info.image
Ptr_Free, info.extra
END

```

Modify the code to add the line in bold, below:

```

IF N_Elements(info) EQ 0 THEN RETURN
Ptr_Free, info.image
Ptr_Free, info.extra
WDelete, info.pixID
END

```

Re-compile your program and run it again now. (If you did not enter the changes above into a file named *histo_gui.pro*, you can use the file *histo_gui.2.pro*, which you downloaded to use with this book. This file must be explicitly compiled to be used. See “The Advantage of Mistakes” on page 261 for more information.)

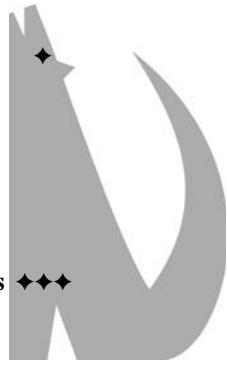
```
IDL> .Compile Histo_Gui  
IDL> Histo_GUI
```

Are the graphics displays smoother? How about when you resize the program?

Good, but there is still much to learn about widget programming in the next chapter. Keep reading.

Chapter 11

♦ Discovering the Possibilities ♦♦♦



Widget Programming Techniques

Chapter Overview

The purpose of this chapter is to demonstrate a few of the most important and frequently used widget programming techniques. Together with the techniques from the previous chapter, these techniques will enable you to write powerful widget programs. Specifically, you will learn:

- How to extend widget program functionality
- How to create pull-down menus
- How to create a program “memory” and an “undo” feature
- How to communicate between widget programs and modules
- How to allow color manipulation transparently on both 8-bit and 24-bit displays
- How to create GIF, JPEG, TIFF, and PostScript files from your graphics display
- How to open a new file from a widget program
- How to send your graphics display directly to the printer

A widget program that cannot be easily extended or added to is a poor widget program, indeed. A primary goal, then, of any widget program we write should be the ability to extend its functionality easily. One of the easiest ways to extend program functionality is to write programs that can call (or be called from) other widget programs. To a large extent this means writing programs in a modular or object-oriented way so that program data and the actions to be performed on the program data are localized in the program itself.

Consider the *Histo_GUI* program you wrote in the last chapter. As it stands, this program is not very functional. In fact, it is really nothing more than a resizeable graphics window for one particular type of display. It would be a better program if it had a bit more functionality. This chapter will show you how to add functionality to this program. (If you haven’t written the program in the last chapter and need a starting point, use the file *histo_gui.2.pro*, which is among the files you downloaded to use with this book. Rename the program file *histo_gui.pro*.)

Adding Image Processing Capability

The first change we would like to make to the *Histo_GUI* program that will make it more useful to us is to add some image processing capability via a pull-down menu. We would like to have a *Processing* button in the menu bar and under that the capability of performing a median and boxcar smoothing operation, a Sobel edge enhancement, and an unsharp masking operation. And, of course, we would like to have some way of returning to the display of the original image.

Building the Pull-Down Menu Widgets

We build a pull-down menu by using the *Menu* keyword on button widget creation routines. Setting the *Menu* keyword makes it possible for that button to be the parent of other button widgets. In practical terms, a button with a *Menu* keyword set for it will not generate an event we can capture in an event handler. Rather, it will allow its child buttons to be seen. In other words, it acts as a pull-down menu.

We are going to add the *Processing* button to the menu bar, but there is no requirement that a pull-down menu come from the menu bar. Any button, anywhere on the interface, can be a pull-down menu button.

Let's insert the pull-down menu code in the *Histo_GUI* widget definition module, just between the creation of the *Quit* button and the draw widget. The original code looks like this:

```
quitID = Widget_Button(fileID, Value='Quit', $  
Event_Pro='Histo_GUI_Quit')  
drawID = Widget_Draw(tlb, XSize=400, YSize=400)
```

We are going to modify it to look like this:

```
quitID = Widget_Button(fileID, Value='Quit', $  
Event_Pro='Histo_GUI_Quit')  
  
processID = Widget_Button(menubarID, Value='Processing', $  
Event_Pro='Histo_GUI_Processing', /Menu)  
  
smoothID = Widget_Button(processID, Value='Smoothing', $  
/Separator, /Menu)  
button = Widget_Button(smoothID, Value='Median Smooth')  
button = Widget_Button(smoothID, Value='Boxcar Smooth')  
  
edgeID = Widget_Button(processID, Value='Edge Enhance', $  
/Menu)  
button = Widget_Button(edgeID, Value='Sobel')  
button = Widget_Button(edgeID, Value='Unsharp Masking')  
  
button = Widget_Button(processID, Value='Original Image')  
  
drawID = Widget_Draw(tlb, XSize=400, YSize=400)
```

Note that the event handler for these buttons is assigned to the *Processing* button with the *Event_Pro* keyword. It is named *Histo_GUI_Processing*. This is the button that is at the top of the pull-down menu in the menu bar. By assigning the event handler to this button, we can capture all of the sub-button events by virtue of the fact that events will “bubble up” the widget hierarchy until they are captured by this event handler.

Note, too, that only the menu buttons (i.e., those with the *Menu* keyword set) have widget identifiers that are unique. All the other buttons have generic “button” identifiers. We obviously will have to “identify” these buttons in the event handler in order to respond appropriately to them. But rather than keeping track of their widget

identifiers, we will identify a button by each button's value. In other words, by the text that is written on the button. The text for each button is, of course, unique.

You can compile and run the program after you make the change. Although be careful not to cause an event. The event handler for the *Processing* button has not been written yet. The pull-down menu should looks similar to the illustration in Figure 110.

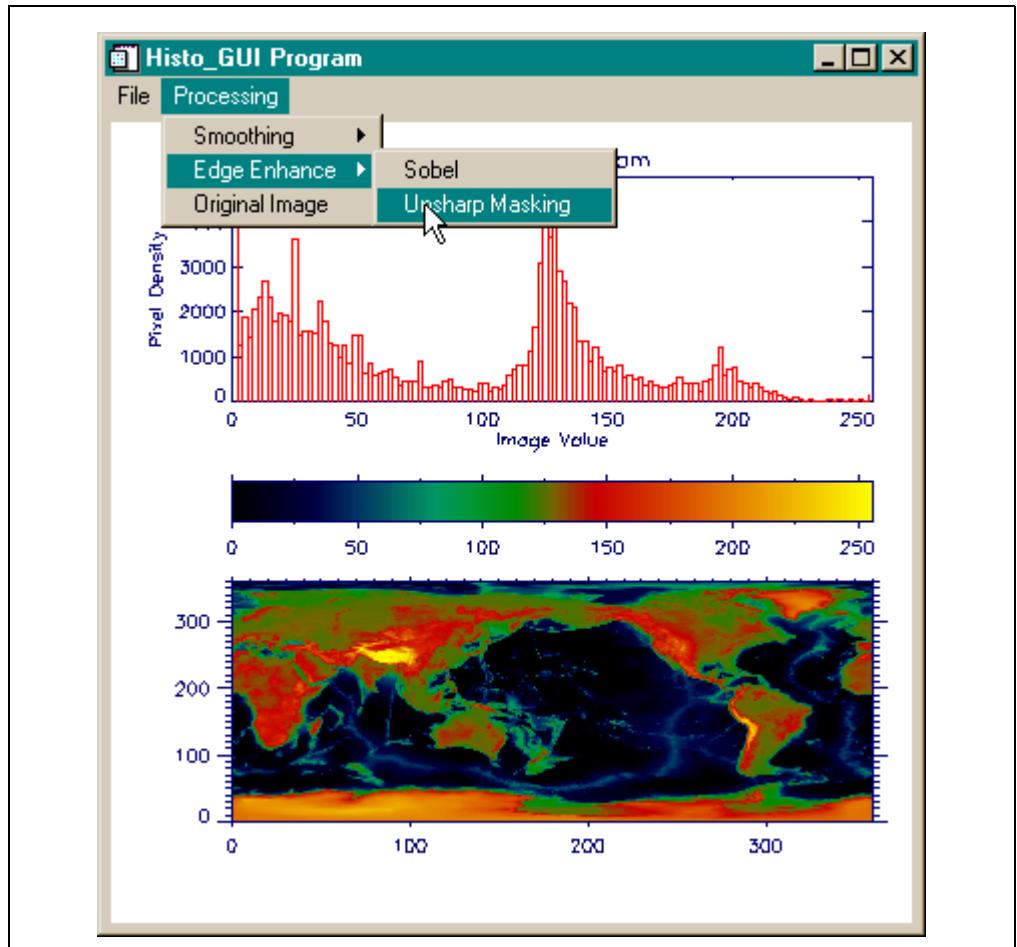


Figure 110: The Histo_GUI program with a Processing pull-down menu added.

Writing the Event Handler for the Pull-Down Menu

We know we are going to have to use the *info* structure in the event handler for the *Processing* button, so we can write the generic outline of the *Histo_GUI_Processing* event handler (in fact, most event handlers) like this. (Add this event handler to your program file somewhere in front of the widget definition module in the file).

```
PRO Histo_GUI_Processing, event
Widget_Control, event.top, Get_UValue=info, /No_Copy
Widget_Control, event.top, Set_UValue=info, /No_Copy
END
```

The real heart of the program module is what happens between the time we get and set the user value of the top-level base. First of all, we need to know which button caused the event, because we have to do something different for each button. We know that the button that caused the event is identified in our event handler as *event.id*. But which button is that?

We can find out by getting the *value* of the button. A button's value is the text that is on the button. In other words, it is exactly the text that we assigned to the button with the *Value* keyword when we created the button. And, since it is text, it is spelled exactly the same way we spelled it when we created it and in the same combination of uppercase and lowercase characters. When we do something with this text, we often want to force it into all uppercase or all lowercase characters with the *StrUpCase* or *StrLowCase* commands.

We could write the code that does the image processing in our event handler like this. Place this code just below the line that gets the *info* structure out of the top-level base and just before the line that sets the *info* structure into the top-level base.

```
Widget_Control, event.id, Get_Value=buttonValue
CASE StrUpCase(buttonValue) OF
    'MEDIAN SMOOTH': processImage = Median(*info.image, 5)
    'BOXCAR SMOOTH': processImage = Smooth(*info.image, 7, $
        /Edge_Truncate)
    'SOBEL': processImage = Sobel(*info.image)
    'UNSHARP MASKING': processImage = Smooth(*info.image, $7) - *info.image
    'ORIGINAL IMAGE': processImage = *info.image
ENDCASE
```

Here the variable *buttonValue* is the value of the button, and we branch on this value in the *CASE* statement that follows.

The only thing left to do is to re-display the graphics, using the *processImage* data rather than the original image data. Type this code just after the code above and just before the line that *Sets* the *info* structure into the user value of the top-level base. Make certain you know which window you are displaying graphics in by making the pixmap widget window the current graphics window with the *WSet* command.

```
WSet, info.pixID
HistoImage, processImage, $
    AxisColorName=info.axisColorName, $
    BackColorName=info.backcolorName, $
    Binsize=info.binsize, $
    DataColorName=info.datacolorName, $
    _Extra=*info.extra, $
    Max_Value=info.max_value, $
    NoLoadCT=1, $
    XScale=info.xscale, $
    YScale=info.yscale

WSet, info.wid
Device, Copy=[0, 0, !D.X_Size, !D.Y_Size, 0, 0, info.pixID]
```

Limitations of the Event Handler as Written

Compile and test the code you have written so far. If it doesn't compile or run correctly, fix the errors and try again.

```
IDL> .Compile histo_gui
IDL> Histo_GUI
```

There are a couple of significant limitations to the program the way it is currently written. Have you discovered them?

First of all, each image processing step is applied to the original image. Most of the time when we are doing image processing, we wish to apply image processing steps

sequentially to the image and have the processing accumulate on the image. This program doesn't allow us to do that.

But there is an even more significant problem. Try this. Start your *Histo_GUI* program and apply the *Unsharp Masking* image processing button. Now resize the graphics window. What happened?

Right. The original image appears in the window instead of the processed image. That is not right at all. When you resize the graphics window, whatever happens to be in the window should be reproduced in the resized window. This program has no—for lack of a better term—*memory* of what has been done to the image. Hence, it can't reproduce it properly.

Both of these problems can be solved by having a second image in the program. I like to call this image the “process” image, because it is the image I am going to display in the graphics window. Modify your *info* structure in the widget definition module to accept a second process image. Of course, initially the process image is identical to the original image data.

```
info = { image:Ptr_New(image), $  
        process:Ptr_New(image), $  
        axisColorName:axisColorName, $  
        backColorName:backcolorName, $  
        binsize:binsize, $  
        dataColorName:datacolorName, $  
        imageColors:imagecolors, $  
        max_value:max_value, $  
        title:title, $  
        xscale:xscale, $  
        yscale:yscale, $  
        extra:Ptr_New(extra), $  
        drawID:drawID, $  
        wid:wid, $  
        pixID:pixID $  
    }
```

With a new pointer in the *info* structure, our *Cleanup* routine will also have to be changed. Locate the *Histo_GUI_Cleanup* module and make this change (in bold) to the code there:

```
ENDIF ELSE BEGIN  
    Ptr_Free, info.image  
    Ptr_Free, info.extra  
    WDelete, info.pixID  
    Ptr_Free, info.process  
ENDELSE
```

Next, modify your *Histo_GUI_Processing* event handler to perform the image processing on the process image and not the original image. Find every instance of the variable *processImage* in the event handler and change it to **info.process*. The code segments, with changes in bold, are shown below:

```
CASE StrUpCase(buttonValue) OF  
    'MEDIAN SMOOTH': *info.process = Median(*info.process, 5)  
    'BOXCAR SMOOTH': *info.process = Smooth(*info.process, 7, $  
        /Edge_Truncate)  
    'SOBEL': *info.process = Sobel(*info.process)  
    'UNSHARP MASKING': *info.process = Smooth(*info.process, $  
        7) - *info.process
```

```
'ORIGINAL IMAGE': *info.process = *info.image
ENDCASE

WSet, info.pixID
HistoImage, *info.process, $
```

Finally, find every instance in your program where you are passing `*info.image` to the `HistoImage` command. There should be two instances currently, both in the `Histo_GUI_TLB_Events` event handler. Both of these instances should be changed so that `*info.image` is changed to `*info.process`. Be sure you do this *twice*.

```
HistoImage, *info.process, $
AxisColorName=info.axisColorName, $
```

Save the program, compile and test it now. Be sure that when you resize the graphics window that what is currently in the window is reproduced faithfully. And you should be able to apply image processing steps sequentially.

Implementing an Undo Capability

The ability to apply image processing steps sequentially causes us to consider the possibility that we may apply a processing step in the wrong order, or that we don't like the result of a processing step. It would be nice if the `Histo_Gui` program had the ability to "undo" a step that was made previously. And possibly "redo" the step if we decide we like it after all.

Such a capability is quite easy to implement in our program, although it requires that we create another "undo" image. In other words, the "undo" image is the previous process image. One can easily implement a multiple undo capability by saving each process image as it changes. A list object of some sort is often used to save this kind of data. You can find a `LinkedList` object on the *Coyote's Guide to IDL Programming* web page that is often used for this purpose.

To implement a single undo/redo capability here, we need to create the *Undo* button. Let's put it at the end of our *Processing* menu. And since it is a different sort of thing from an image processing operation, let's add a separator or line above the item to distinguish it as something different. We can do this by setting the *Separator* keyword.

Find the line creating the *Original Image* button in your widget definition module and add the following code (in bold) just after it:

```
button = Widget_Button(processID, Value='Original Image')
undoID = Widget_Button(processID, Value='Undo', $
UValue='Redo', Event_Pro='Histo_GUI_Undo', $
Sensitive=0, /Separator)
```

Note that we place the text "Redo" in the user value of the button. We will switch between the value text and the user value text as we work with the button. Also note that the button is created in an insensitive (`Sensitive=0`) condition. This will cause it to be grayed out or unavailable in the interface. This is appropriate when the program first starts up because there has been no processing step to "undo" yet. We will make the button sensitive after we perform an image processing operation.

Finally, note that we attach a separate event handler to this button, named `Histo_GUI_Undo`. We don't have to have a separate event handler, of course. We could process this event in the `Histo_GUI_Processing` event handler along with the events from the other buttons beneath the `Processing` button on the menu. But since this operation is different from those, a separate event handler makes sense.

Next, modify the `info` structure to include an *undo* image and the *Undo* button identifier. The code will look like this, with the changes in bold type:

```

info = { image:Ptr_New(image), $
process:Ptr_New(image), $
undo:Ptr_New(image), $
undoID:undoID, $
axisColorName:axisColorName, $

```

Next, modify the *Histo_GUI_Cleanup* routine to free the undo image pointer. Make the changes in bold type:

```

Ptr_Free, info.image
Ptr_Free, info.extra
WDelete, info.pixID
Ptr_Free, info.process
Ptr_Free, info.undo

```

The next step is to modify the *Histo_GUI_Processing* event handler. The idea here is that whenever we enter the image processing operation event handler we should make sure the *Undo/Redo* button is turned on and that it says *Undo*. We should also save the current process image in the undo image pointer. Add the two lines in bold in the code below near the front of the *Histo_GUI_Processing* event handler:

```

PRO Histo_GUI_Processing, event
Widget_Control, event.top, Get_UValue=info, /No_Copy
*info.undo = *info.process
Widget_Control, info.undoID, Set_Value='Undo', $
    Set_UValue='Redo', Sensitive=1
Widget_Control, event.id, Get_Value=buttonValue

```

Finally, we are ready to write the *Histo_GUI_Undo* event handler. We start by writing the foundation of almost every event handler.

```

PRO Histo_GUI_Undo, event
Widget_Control, event.top, Get_UValue=info, /No_Copy
Widget_Control, event.top, Set_UValue=info, /No_Copy
END

```

Now we fill in what we want the event handler to do. First, we want to switch the process and undo images. The code looks like this (in bold):

```

PRO Histo_GUI_Undo, event
Widget_Control, event.top, Get_UValue=info, /No_Copy
temp = *info.process
*info.process = *info.undo
*info.undo = temp

```

Next, we want to make the text on the *Undo* button read *Redo*. Or, visa versa, if we are doing a redo instead of an undo. In any case, we can accomplish this just by switching the text strings in the user value and value of the button. The code, placed directly below the code you just wrote, looks like this:

```

Widget_Control, event.id, Get_Value=theValue, $
    Get_UValue=theUValue
Widget_Control, event.id, Set_Value=theUValue, $
    Set_UValue=theValue

```

And, finally, all we have to do is re-draw the graphic display with the process image. Place this code (in bold) directly below the code you just wrote and before the last two lines in the module:

```

WSet, info.pixID
HistoImage, *info.process, $
AxisColorName=info.axisColorName, $

```

```

BackColorName=info.backcolorName, $
Binsize=info.binsize, $
DataColorName=info.datacolorName, $
_Extra=*info.extra, $
Max_Value=info.max_value, $
NoLoadCT=1, $
XScale=info.xscale, $
YScale=info.yscale

WSet, info.wid
Device, Copy=[0, 0, !D.X_Size, !D.Y_Size, 0, 0, info.pixID]

Widget_Control, event.top, Set_UValue=info, /No_Copy
END

```

Save the file, recompile your program, and test it. (If you did not enter the program above into a file named *histo_gui.pro*, you can use the file *histo_gui.3.pro*, which you downloaded to use with this book. This file must be explicitly compiled to be used. See “The Advantage of Mistakes” on page 261 for more information.)

Does the *Undo/Redo* button work as it is suppose to? The program with the *Undo* button should look similar to the illustration shown in Figure 111.

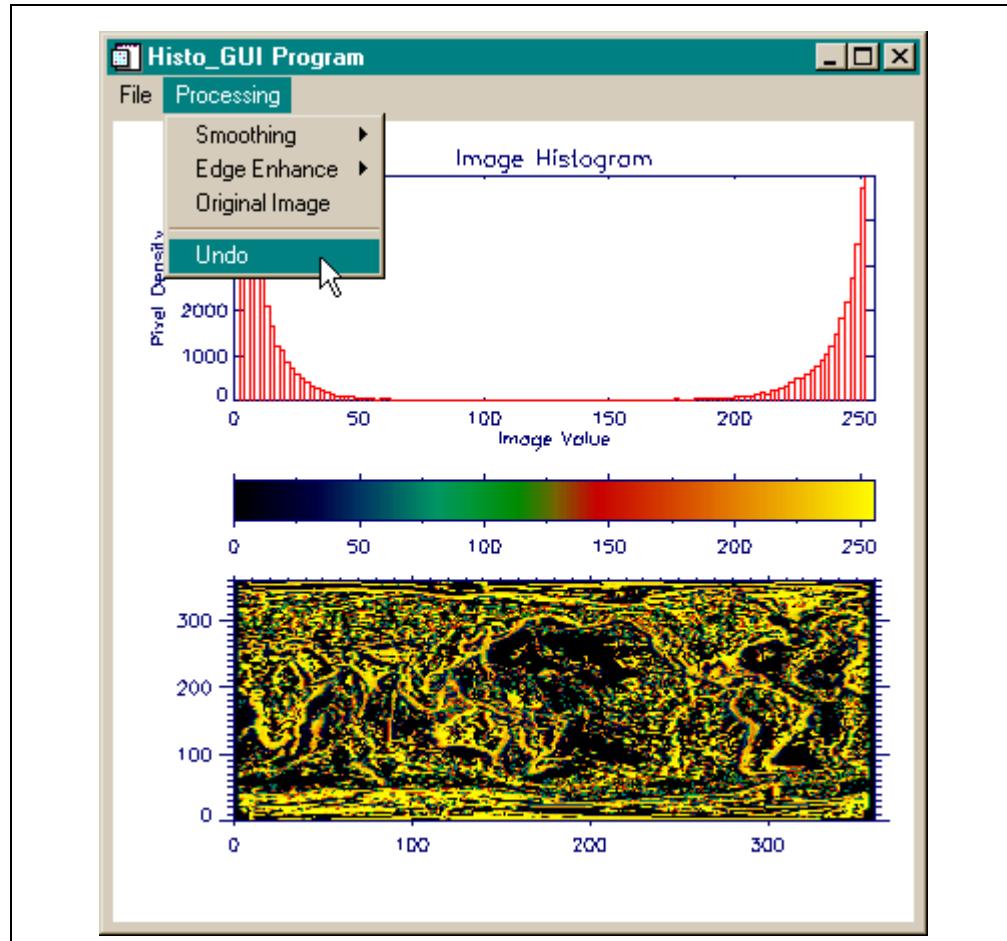


Figure 111: The *Histo_Gui* program with Unsharp Masking applied and the *Undo* button ready to reverse the processing step.

Adding Color Controls to the Program

The next step in program development is to add the capability of controlling the programs colors from the graphical user interface. In the *HistoImage* program, there are the three drawing colors, which we are asking for by name, and the image colors, which we wish to control externally.

Building the Pull-Down Menu Widgets

We can create another pull-down menu in the menu bar for this purpose. In this case, we will assign a separate event handler for the image colors and another one to handle the drawing colors. We will use the user value of the drawing color widgets to determine which button actually caused the event. The code, which you should place in your widget definition module just after the creation of the *Undo* button and just before the creation of the draw widget, will look like this. Changes are in bold type.

```

undoID = Widget_Button(processID, Value='Undo', $  

    UValue='Redo', Event_Pro='Histo_GUI_Undo', $  

    Sensitive=0, /Separator)  
  

colorsID = Widget_Button(menuBarID, Value='Colors', /Menu)  

button = Widget_Button(colorsID, Value='Image Colors', $  

    Event_Pro='Histo_GUI_Image_Colors')  

drawColorsID = Widget_Button(colorsID, /Menu, $  

    Value='Drawing Colors', $  

    Event_Pro='Histo_GUI_Drawing_Colors')  

button = Widget_Button(drawColorsID, Value='Data Color', $  

    UValue='DATA')  

button = Widget_Button(drawColorsID, $  

    Value='Background Color', UValue='BACKGROUND')  

button = Widget_Button(drawColorsID, $  

    Value='Annotation Color', UValue='ANNOTATION')  
  

drawID = Widget_Draw(tlb, XSize=400, YSize=400)

```

Note that the event handler for the drawing colors is assigned to the *Drawing Colors* button, which cannot generate its own events, since it is a menu button. Events from this button's children widgets will bubble up and be captured by this event handler. You will be able to determine which button it is that is generating the event by examining that button's user value.

Writing the Drawing Colors Event Handler

Creating the initial *Histo_GUI_Drawing_Colors* event handler code should be second nature to you by now. Create the event handler somewhere in the program file ahead of the widget definition module. Add these four lines:

```

PRO Histo_GUI_Drawing_Colors, event  

Widget_Control, event.top, Get_UValue=info, /No_Copy  

Widget_Control, event.top, Set_UValue=info, /No_Copy  

END

```

The first order of business is to determine which of the three possible buttons caused the event. In a previous event handler we looked at the button's value. In this event handler, we look at the button's *user* value. One technique is not any better than the other, but it is good to know about both of them, because you will see both in IDL code you are trying to understand.

To get the button's user value, type this line of code directly after you get the *info* structure out of its storage location:

```
Widget_Control, event.id, Get_UValue=buttonUValue
```

You will branch on this value in a *CASE* statement.

What we plan to do in this event handler is call a program named *PickColorName* that is among the program files you downloaded to use with this book. *PickColorName* is what I like to call a *modal dialog form widget*. I mean by this that the program is a widget program that can be called from within another widget program to gather information from the user (usually on some kind of form or graphical layout) and return that information to the caller. This information will be used in the calling program when the form widget is destroyed. In this case, the information we are looking for is the name of a new color to use. This will be returned to you when the form widget is destroyed, either by clicking the *Cancel* or the *Accept* button. You will learn how to write dialog form widgets in the next chapter.

For now it is enough to know that the *PickColorName* program is written as a modal dialog widget. That is to say, this program will stop all other programs from accepting user input until it is destroyed. This includes the *Histo_GUI* program it was called from. The purpose of a modal widget is to stop everything until some information is collected from the user, and then continue program execution.

Modal widgets, or widgets that stop all other user interaction, cannot be created without the presence of a *group leader*. A group leader in IDL is a widget, typically the top-level base widget of the calling program. The idea is that if the group leader is destroyed, all members of that group should be destroyed as well. You will learn more about group leaders in just a moment. But you should be aware that in this program the only way to be certain *PickColorName* will be a modal widget is to pass it a group leader.

There is only one positional parameter to the *PickColorName* program, *theName*, which is the name of the color *PickColorName* will start with. The names *PickColorName* is familiar with are the same color names you use with *GetColor*. For example, names like: *Black*, *Magenta*, *Cyan*, *Yellow*, *Green*, *Red*, *Blue*, *Navy*, *Pink*, *Aqua*, *Orchid*, *Sky*, *Beige*, *Charcoal*, *Gray*, and *White*. To see a complete list of names, type this:

```
IDL> Print, GetColor(/Names)
```

If you mis-spell a name, or ask for a name that is not recognized, or don't include a name, you start with a white color. For example, try this at the IDL command line:

```
IDL> colordname = PickColorName()
```

You should see a program that looks similar to the illustration in Figure 112.

The return value of the function is the name of the selected color. Try selecting a different color by clicking on the different colors in the first two color rows. The name of the color and the color itself are loaded in the color patch in the middle of the program window.

```
IDL> Print, colordname  
Beige
```

If you wanted to start *PickColorName* with a red color, for example, you type this:

```
IDL> colordname = PickColorName('red')
```

In addition to the one positional parameter, *PickColorName* is defined with five keyword parameters:

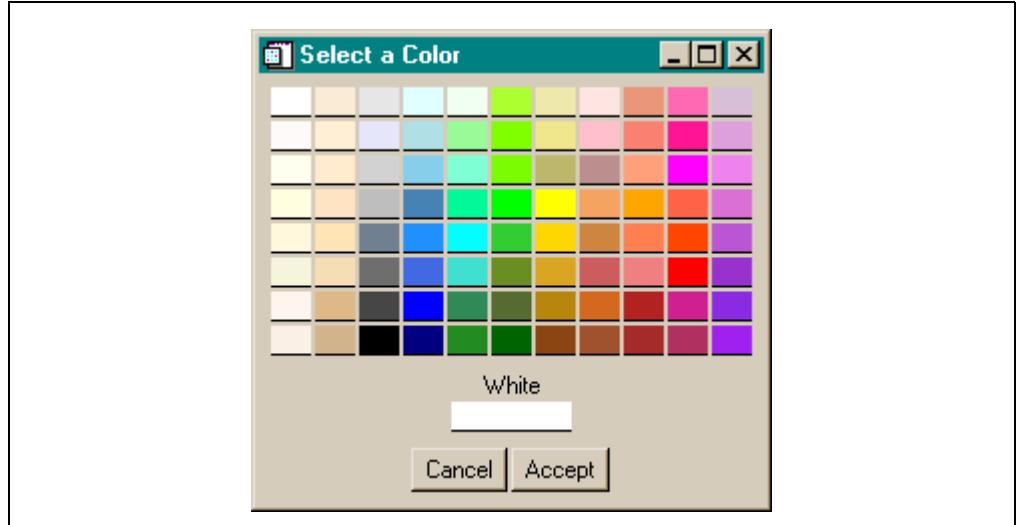


Figure 112: The *PickColorName* dialog form widget. Users select one of 88 available colors with the mouse. The name of the color is returned as the result of the function.

Bottom

In order to see colors, *PickColorName* has to load them in the color table. This keyword selects the index where those colors are loaded. The “mixing” or “current” color is loaded at the *Bottom* index, and the 16 predefined colors are loaded at the following indices. If *Bottom* is not specified, the default is to use *!D.Table_Size - 18*.

Cancel

This is an output keyword. When the program returns, this keyword value will be set to 1 if the user selected the *Cancel* button, killed the widget with the mouse, or if a program error occurred. It will be set to 0 if the user selected the *Accept* button.

Group_Leader

This keyword identifies the group leader for the *PickColorName* program. When the group leader is destroyed, all members of that group will be destroyed also. The *Group_Leader* keyword is required for proper modal widget behavior. In that sense, the *Group_Leader* keyword is a required keyword if the *PickColorName* program is being called from within a widget program.

Index

This optional keyword identifies a color table index number that should be set to the selected color upon exiting the *PickColorName* program. In other words, not only can you obtain the name of the selected color, but you can load the color in the color table by using this *Index* keyword. If the keyword is not used, no color is loaded.

Title

This keyword selects a title for the *PickColorName* top-level base. The title is set to “Select a Color” if no text is provided.

With this information in mind, we can write the *CASE* statement for the event handler. It will be written like this:

```
CASE buttonUValue OF
  'ANNOTATION': BEGIN
    colorname = PickColorName(info.axisColorName, $
      Cancel=cancelled, Group_Leader=event.top, $
      Title='Select Annotation Color', $
```

```

        Index=!D.Table_Size-2, Bottom=!D.Table_Size-21)
    IF NOT cancelled THEN info.axisColorName = colortname
    END
'DATA': BEGIN
    colortname = PickColorName(info.dataColorName, $
        Cancel=cancelled, Group_Leader=event.top, $
        Title='Select Data Color', $
        Index=!D.Table_Size-3, Bottom=!D.Table_Size-21)
    IF NOT cancelled THEN info.dataColorName = colortname
    END
'BACKGROUND': BEGIN
    colortname = PickColorName(info.backColorName, $
        Cancel=cancelled, Group_Leader=event.top, $
        Title='Select Background Color', $
        Index=!D.Table_Size-4, Bottom=!D.Table_Size-21)
    IF NOT cancelled THEN info.backColorName = colortname
    END
ENDCASE

```

Notice that in each case, we are passing the current drawing color to the *PickColorName* function and checking to see if the user selected the *Cancel* button. If they did not cancel, we are storing the returned color name into the appropriate field of our *info* structure. We are also making the *Histo_GUI* top-level base be the group leader for the *PickColorName* program. Remember this is required if we want *PickColorName* to be a modal program.

We are choosing to load the actual color table value in this case, since we are using the *Index* keyword. You probably had to check the *HistoImage* program to see where these color values were actually loaded to get the appropriate value for the *Index* keyword.

Notice that we chose to load the 17 colors from the *PickColorName* program at color index *!D.Table_Size-21* with the *Bottom* keyword. It really doesn't matter where we load these colors if we are running the program on a 24-bit display, but if we are running the program on an 8-bit display, we don't want the colors interfering with the drawing colors we have already loaded. So we load them somewhere below our drawing colors in the color table. Of course, what this means is that when the *PickColorName* program is on the display, some of the image colors will be incorrect. But since *PickColorName* restores the color table that it encounters when it first starts up, these colors will be repaired when *PickColorName* exits. The only color that will be changed is the color indicated by the *Index* keyword.

But, since that *Index* color *did* change, the next thing to do is obtain the color table vectors and store them in the appropriate fields of the *info* structure. We can use the *TVLCT* command with the *Get* keyword set to obtain the current color table vectors. The code will look like this:

```

TVLCT, r, g, b, /Get
info.r = r
info.g = g
info.b = b

```

Normally when we make a change to our program, especially when that is a color change, we must re-display the graphics. But we are not doing so here. Can you think of a reason why it is not necessary in this case?

The reason is that we have already set up keyboard focus events to load the color tables and re-display the graphics (if needed) for us. When we exit the *PickColorName* program, the keyboard focus will return to the *Histo_GUI* program

that called *PickColorName* and the appropriate action will take place. It is a consequence of the way we have written our color protection scheme.

Test the program by saving, compiling, and running it. Can you change the background and data colors? Does it work the way you expect it do?

Writing the Image Colors Event Handler

The next step is to write the event handler for controlling the image colors. As before, we write the event handler skeleton code like this. Place it somewhere ahead of the widget definition module in your program file.

```
PRO Histo_GUI_Image_Colors, event
Widget_Control, event.top, Get_UValue=info, /No_Copy
Widget_Control, event.top, Set_UValue=info, /No_Copy
END
```

In the drawing colors event handler you just wrote you called a *modal* dialog form widget. The fact that it was modal meant that program execution stopped in the event handler until the *PickColorName* program was destroyed. This is a perfectly acceptable way to operate programs, but sometimes it is inconvenient. For example, many people like to have a color table changing tool on the display simultaneously with their graphics display. They want it there in case they decide to change the colors in their graphics display. But they don't want to be forced to change the colors to use their other programs.

So to accommodate these *non-modal* dialog form widgets, some other way of communicating between widget programs is required. The reason such communication is important is because certain actions must be taken when new colors are loaded into the color table. For example, on 8-bit and 24-bit display devices the new color vectors must be loaded into the *info* structure. On 24-bit display devices the graphics must be redisplayed to take advantage of the newly loaded colors. But—and this is the critical piece of information we don't have and need—we have to know when the new color table vectors were actually loaded.

If you called a program like *XLoadCT* or *XColors* (which is a program that you downloaded to use with this book) from the event handler, the program could remain on the display for a long time before the user got around to using it. How would you know when the color vectors were loaded?

The answer, quite frankly, is you wouldn't. At least, not unless the program that loaded the color table vectors could somehow communicate to the program that called it that the color vectors had been loaded.

You have already seen that both *XLoadCT* and *XColors* have the ability to notify a procedure when they load the color tables. (See “Automatic Updating of Graphic Displays When Color Tables are Loaded” on page 66.) But while this notification strategy works reasonably well for many simple programs and from the IDL command line, it is much harder to make it work in a widget program. And the reason this is so, is because of the delay in action that may be involved.

The procedure notification method involves copying some sort of data into either the *XLoadCT* or *XColors* program that can be used later when the color vectors are loaded into the color table. We can imagine how we should write the procedure and what kind of data we would need. In fact, the procedure will probably have to re-draw the graphics display (certainly it will on 24-bit displays), but it will absolutely require access to information that currently resides in the *info* structure.

There are two problems with passing the *info* structure to the procedure. First, we have been studiously trying to avoid copying the *info* structure at all. Second, if we made a

copy of the *info* structure, and the user delayed loading the color table, we can imagine that the information in the copied *info* structure (say, for example, **info.process*, the process image) would be out of date by the time the procedure was called into action.

A likely scenario would be to get the color table tool on the display, change the color table, change the background color with the *PickColorName* program, and then resize the image. The background color would be the color of the graphics display when the color table tool was called, not the current background color.

Both of these problems can be solved by pointers. We can either make each field in the *info* structure a pointer, or we can make a pointer to the *info* structure itself, and pass this to the color table tool. But this seems like a lot of work to me. And, quite frankly, you are not going to find many IDL programs written this way. What happens when you want to add a color table tool to those programs?

So, I prefer to communicate to other widget programs in a way that seems more natural to me in a widget program. I prefer to communicate by means of widget events.

Since communicating via widget events is not possible with the IDL-supplied *XLoadCT* program, we are going to use *XColors* as the color table tool in the *Histo_GUI* program. (You will learn more about communicating via widget events in “Creating a Non-Modal Widget Dialog” on page 340.)

Communicating in *XColors* via Widget Events

The way the widget event notification for *XColors* is implemented is via a *NotifyID* keyword. This keyword accepts a two-element array that specifies the widget to be notified when a color table is loaded, and the widget at the top of the former widget's hierarchy. (The keyword can actually accept a 2-by-*n* array of widget identifiers, if there is more than one widget you wish to notify.) In pseudo code, this might look like this:

```
XColors, NotifyID=[widgetToNotify, WTN_TLB]
```

Where *widgetToNotify* is the identifier of a widget to notify when the color table is loaded, and *WTN_TLB* is the top-level base widget identifier in the *widgetToNotify*'s widget hierarchy.

At the time *XColors* loads the color table vectors, it checks to see if there is a widget to notify and if that widget is still a valid widget identifier. If this is the case, *XColors* builds an event structure defined like this:

```
xcolorsEvent = { XCOLORS_LOAD, $  
                 ID: widgetToNotify, $  
                 Top: WTN_TLB, $  
                 Handler: 0L, $  
                 R: r, $  
                 G: g, $  
                 B: b, $  
                 Index: ct_index, $  
                 Name: ct_name  
               }
```

The event structure has the name *XCOLORS_LOAD*. The *ID* field is set equal to the identifier of the widget to notify. The *Top* field is set equal to the widget at the top of the *widgetToNotify* hierarchy. The *Handler* field value is unknown at the time the event structure is created, but it is set equal to a long integer. IDL will actually fill in the correct value for this field. in the process of sending this event to the correct widget. The *R*, *G*, and *B* fields have the color vectors that were just loaded into the color table (via *TVLCT*). The *Index* field contains index number of the currently

loaded color table. And the *Name* field contains the name of the currently loaded color table.

Once the event structure is created, it is sent to the *widgetToNotify* widget by means of the *Send_Event* keyword to the *Widget_Control* command, like this:

```
Widget_Control, widgetToNotify, Send_Event=xcolorsEvent
```

This *xcolorsEvent* event structure is just like any other event structure in IDL. That is to say, it goes through the same process of getting into the event queue to be processed in order, etc. IDL will check to see, as it puts the event structure on the event queue, which widget is associated with the event handler for the *widgetToNotify* widget, and will fill out the *Handler* field of the event structure with the proper widget identifier. Of course, the event handler that will receive this event structure will have to be written in a way that anticipates receiving an event structure like this.

In practice, the *widgetToNotify* widget is usually the same button widget that was selected to get you into the event handler that called *XColors* in the first place. That is to say, usually we call *XColors* like this:

```
XColors, NotifyID=[event.id, event.top]
```

For example, in the *Histo_GUI* program, *event.id* would refer to the *Image Colors* button. And *event.top* would be the top-level base identifier for the *Histo_GUI* program.

With this in mind, we can start to fill out the *Histo_GUI_Image_Colors* event handler. The first order of business is to determine what kind of an event has come into the event handler. It could either be a *WIDGET_BUTTON* event structure (if it comes from selecting the *Image Colors* button), or it could be this new *XCOLORS_LOAD* event structure (if *XColors* loaded a color table). It is possible to distinguish between the two by using the *Tag_Names* command with the *Structure_Name* keyword set to return the structure name. So the first line of code in the *Histo_GUI_Image_Colors* event handler after the *info* structure is obtained is this (in bold type):

```
Widget_Control, event.top, Get_UValue=info, /No_Copy
thisEvent = Tag_Names(event, /Structure_Name)
```



Remember that *Tag_Names* returns uppercase string variables. This will be critical in the *CASE* statement that follows.

On the following line, we can add the skeleton of the *CASE* statement, so that the rest of the event handler looks like this:

```
CASE thisEvent OF
  'WIDGET_BUTTON': BEGIN
    END
  'XCOLORS_LOAD': BEGIN
    END
  ENDCASE
Widget_Control, event.top, Set_UValue=info, /No_Copy
END
```

Now it is just a matter of filling out what should be done for each case. In the button case, of course, we want to call the *XColors* program.

Another feature of the *XColors* program that makes it more attractive to use in widget programs than the IDL-supplied *XLoadCT* program, is that it is possible to have multiple copies of the program on the display at the same time. This is not possible in

XLoadCT, because that program stores program information in a common block. Widget programs that have common blocks must protect those common blocks by making sure no more than one version of the program can ever appear on the display at the same time.

Protecting common blocks in widget programs is done with the *XRegistered* command, which checks to see if a program with some specified name is registered with *XManager*. The *XRegistered* command will return a 0 if no program with that name is registered. It will return something other than a 0 if there is a program with that name registered. For example, type these commands at the IDL command line:

```
IDL> running = XRegistered('xloadct') & Print, running
      0
IDL> XLoadCT
IDL> running = XRegistered('xloadct') & Print, running
      1
```

Notice that *XRegistered* has the nice effect of bringing the program forward on the display if it is already registered.

Programs with common blocks can protect those common blocks by having code similar to this line from *XLoadCT* in the first couple of lines of the widget definition module, before any widgets are defined:

```
IF XRegistered('xloadct') NE 0 THEN RETURN
```

But, as I say, *XColors* doesn't suffer from this limitation, since it doesn't use common blocks. (Nor do any of the widget programs you downloaded to use with this book. I write *all* of my widget programs without common blocks.) *XColors* does have a limitation that you can only have one *XColors* with a particular title on the display at any one time. I impose this limitation because I don't want *XColors* programs proliferating like rabbits every time you click an *Image Colors* button!

But, I would like to have a different *XColors* program associated with each *Histo_GUI* program I wanted to run. That way I could change the colors in each *Histo_GUI* program independently of any others. To do that, I need to give the *XColors* program a unique title in each *Histo_GUI* program I run.

Humm. How can I do that? What is there that is unique in each *Histo_GUI* program?

Actually, there are many things unique about each *Histo_GUI* program. All the widget identifiers are unique, for example. But I often prefer to use the window index number, since that is a nice simple, two-digit number. And if I use the two-digit number on both the *Histo_GUI* program window and the *XColors* window, then I have a way of telling which *XColors* program changes colors for which *Histo_GUI* window.

We are almost ready to write the event handler code, I promise. Just one more point.

When *XColors* (or *XLoadCT* for that matter) is invoked, it has no way of knowing what has occurred to the color table vectors prior to that time. It simply gets the color table vectors that are currently loaded and uses those as the starting colors. If you want these starting colors to reflect the colors in the *Histo_GUI* window, you will have to load those color vectors before invoking the *XColors* program.

Alright, then. Here is the code for the *WIDGET_BUTTON* case. New code is added in bold characters:

```
'WIDGET_BUTTON': BEGIN
    TVLCT, info.r, info.g, info.b
    colorTitle = info.title + " (" + StrTrim(info.wid,2) + ")"
    Widget_Control, event.top, TLB_Set_Title = colorTitle
```

```

XColors, NColors=info.imagecolors, $
NotifyID=[event.id, event.top], $
Title=colortitle + ' Colors'

END

```

Notice the title we are putting on the top-level base widget is composed of the original title with the window index number set inside of parentheses. The *StrTrim* command is used to convert the window index number to a string, while at the same time trimming blank characters from both ends of the string. (This is the meaning of the number 2 in the *StrTrim* command.) String concatenation occurs by means of the plus (“+”) operator in IDL.

Notice, too, that the *XColors* program is restricted to changing just the image colors and not the data colors in the color table by the use of the *NColors* keyword and the *info.imagecolors* value.

Next, we are ready to write the code for the *XCOLORS_LOAD* case. What do we want to do when a different color table is loaded? See the new colors in the *Histo_GUI* display, obviously. What is required to do that? Just two things: (1) we need to save the new color table vectors in the appropriate fields of the *info* structure, and (2) we need to re-draw the graphics display if we are on a 24-bit display device.

Notice the difference between what we have to do here and what we had to do in the *Histo_GUI_Drawing_Colors* event handler. There we could rely on the keyboard focus event generated when the *PickColorName* program was destroyed to re-draw the graphics display, here we have to do it ourselves. The reason for this is that the focus may not be returning to the *Histo_Gui* program immediately. Since *XColors* is a non-modal widget dialog, the keyboard focus may remain on the *XColors* program. But we still wish to see the new colors reflected in the *Histo_GUI* display window, no matter where the keyboard focus is located.

The code for the *XCOLORS_LOAD* case will look like this. Add the code in bold characters:

```

'XCOLORS_LOAD': BEGIN
    info.r = event.r
    info.g = event.g
    info.b = event.b
    Device, Get_Visual_Depth=theDepth
    IF theDepth GT 8 THEN BEGIN
        WSet, info.pixID
        HistoImage, *info.process, $
            AxisColorName=info.axisColorName, $
            BackColorName=info.backcolorName, $
            Binsize=info.binsize, $
            DataColorName=info.datacolorName, $
            _Extra=*info.extra, $
            Max_Value=info.max_value, $
            NoLoadCT=1, $
            XScale=info.xscale, $
            YScale=info.yscale
        WSet, info.wid
        Device, Copy=[0,0,!D.X_Size,!D.Y_Size,0,0,info.pixID]
    ENDIF
END

```

Save your program, re-compile it, and run it now.

```
IDL> .Compile histo_gui
IDL> Histo_Gui
```

Try changing the image colors. Your program should look similar to the illustration in Figure 113. What happens if you have several *Histo_GUI* programs running at the same time. Can you control each window's colors independently?

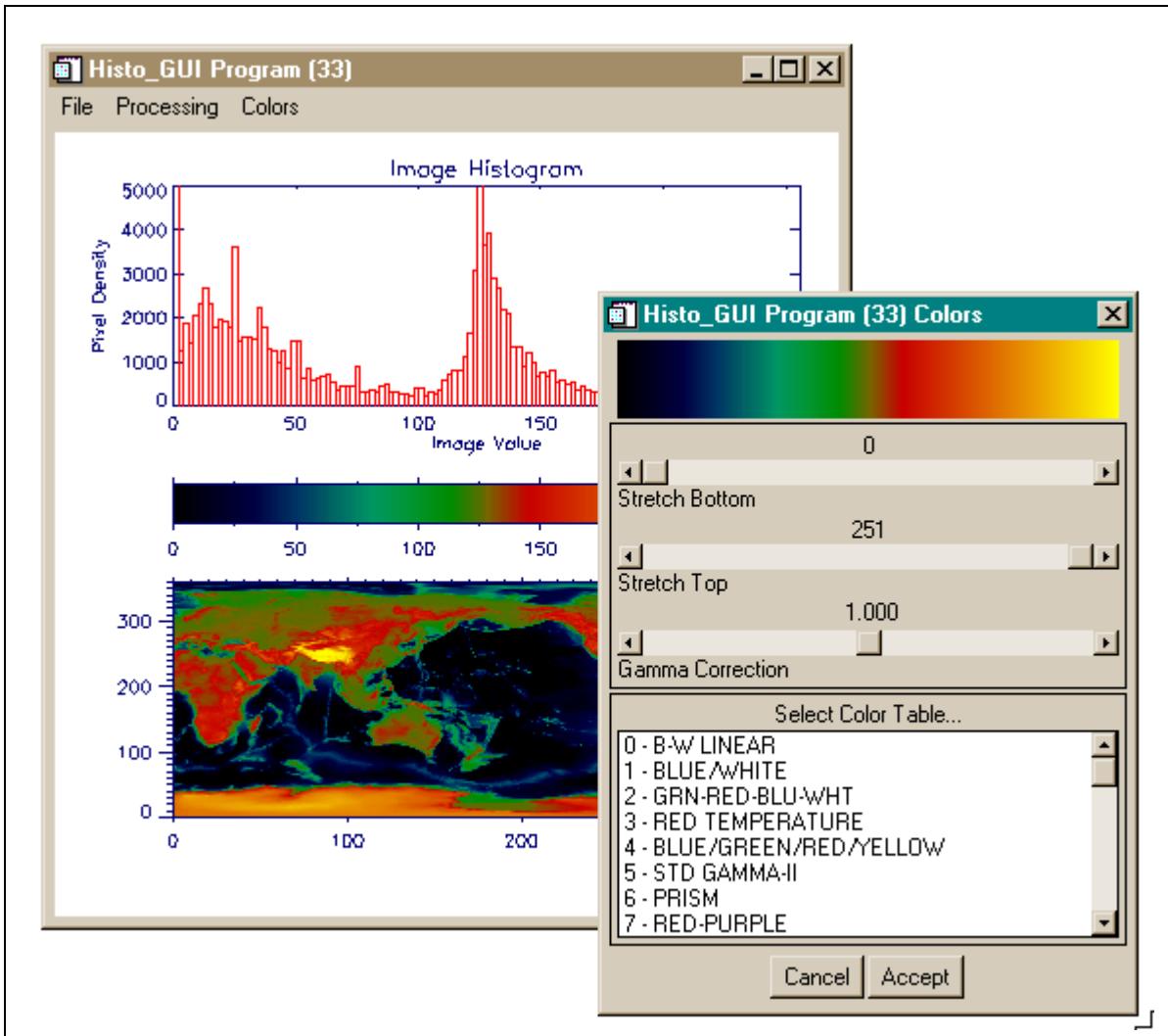


Figure 113: The *Histo_GUI* program and the *XColors* program on the display at the same time. Notice the titles on the two programs.

Importance of Group Leaders

There is just one minor problem with this program and the way it is currently written.

Try this. Get the *Histo_GUI* program and the *XColors* program on the display simultaneously by selecting the *Image Colors* button under the *Colors* menu item. When you have both programs on the display, select the *Quit* button under the *File* menu on the *Histo_GUI* program. What happens?

The *Histo_GUI* program disappears, but the *XColors* program stays on the display. This is not exactly what you want to have happen. In some sense, the *XColors* program “belongs” to the *Histo_GUI* program. You might think that the *XColors* program was “spawned” from the *Histo_GUI* program. In any case, it is good

programming practice to clean up after yourself. What you would like it have happen is to have the *XColors* program disappear when the *Histo_GUI* program disappears.

This kind of behavior is easily accomplished with widgets by making one widget the group leader of another. When the group leader widget is destroyed, all of the members of that group are also destroyed. In this case, we would like to make the *Histo_GUI* top-level base widget be the group leader for the *XColors* program.

As it happens, the *XColors* program already has a *Group_Leader* keyword defined for exactly this purpose. All you have to do to take advantage of it, is pass the identifier of the group leader. In this case, that value is in the variable *event.top*. Modify the lines in the *Histo_GUI_Image_Colors* event handler in which *XColors* is invoked by adding the code in bold below:

```
XColors, NColors=info.imagecolors, $  
    NotifyID=[event.id, event.top], $  
    Title=colortitle + ' Colors', $  
Group_Leader=event.top
```

Close any *XColors* program you currently have on your display. Re-compile the *Histo_GUI* code after making this change, and select the *Image Colors* button. Now when you quit the *Histo_GUI* program, the *XColors* program should also be destroyed.

This ability to make a widget program part of another widget group is so powerful that it is doubtful that we would write a widget program without a *Group_Leader* keyword defined for it. For example, we might want to use *Histo_GUI* functionality in some other widget program we are writing. If we can get the *Histo_GUI* program to be destroyed when its group leader is destroyed, then we can use its functionality any time we need it.

It is so trivially easy to make a program respond to a group leader's death, that we should do it as a matter of course on all widget program we write. The first step is to define a *Group_Leader* keyword for the program. This is done on the procedure definition statement of the widget definition module. Make the following addition (in bold characters) to the *Histo_GUI* code:

```
PRO Histo_GUI, $ ; The program name.  
    image, $ ; The image data.  
    AxisColorName=axisColorName, $ ; The axis color.  
    BackColorName=backcolorName, $ ; The background color.  
    Binsize=binsize, $ ; The histogram bin size.  
    ColorTable=colortable, $ ; The colortable index.  
    DataColorName=datacolorName, $ ; The data color.  
    _Extra=extra, $ ; Extra keywords.  
Group_Leader=group_leader, $ ; The group leader.  
    Max_Value=max_value, $ ; The histogram max value.  
    Title=title, $ ; The program title.  
    XScale=xscale, $ ; The X image scale.  
    YScale=yscale ; The Y image scale.
```

It is not even necessary to check this keyword to see if it's value is defined, like we do for most input keywords. Rather, we can pass the *group_leader* variable along directly and assign it as the group leader for the top-level base widget by using the *Group_Leader* keyword on the *XManager* command. Find the *XManager* command at the end of the *Histo_GUI* widget definition module (the program module you just modified). Make the following change (in bold) to the code:

```
XManager, 'histo_gui', tlb, /No_Block, $  
    Event_Handlers='Histo_GUI_TLB_Events', $  
    Cleanup='Histo_GUI_Cleanup', Group_Leader=group_leader
```

Save your file, re-compile, and test the program. (If you did not enter the program above into a file named *histo_gui.pro*, you can use the file *histo_gui.4.pro*, which you downloaded to use with this book. This file must be explicitly compiled to be used. See “The Advantage of Mistakes” on page 261 for more information.)

```
IDL> .Compile histo_gui  
IDL> Histo_GUI
```

Does the program work as you expect it to? Good. Then let’s add even more functionality.

Adding File Output Functionality

The next piece of functionality we would like to add to the *Histo_GUI* program is the ability to save the graphics display in various file output formats. The formats I am going to show you how to create are GIF, JPEG, TIFF and PostScript formats. I choose these because if you know how to do these, you know how to do most of the file output formats IDL provides, with the exception of the scientific file formats (HDF, CDF, and netCDF).

Building the Pull-Down Menu Widgets

The first step is to add the proper buttons to the *Histo_GUI* interface. A good place to put the buttons is between the *File* menu button and the *Quit* button in the *File* pull-down menu. Let’s put these formats under a *Save As* button.

Modify the *Histo_GUI* widget definition module code by make the additions in bold in the code below:

```
fileID = Widget_Button(menuBarID, Value='File', /Menu)  
  
saveAsID = Widget_Button(fileID, Value='Save As', $  
    Event_Proc='Histo_GUI_File_Output', /Menu)  
button = Widget_Button(saveAsID, Value='GIF File', $  
    UValue='.gif')  
button = Widget_Button(saveAsID, Value='JPEG File', $  
    UValue='.jpg')  
button = Widget_Button(saveAsID, Value='TIFF File', $  
    UValue='.tif')  
button = Widget_Button(saveAsID, Value='PostScript File', $  
    UValue='.ps')  
  
quitID = Widget_Button(fileID, Value='Quit', $  
    Event_Proc='Histo_GUI_Quit', /Separator)
```

Notice we name a new event handler, *Histo_GUI_File_Output*, to handle the events from all four of the new file output buttons. And notice that we separate the *Save As* button from the *Quit* button (which has a very different effect), by adding a *Separator* keyword to the *Quit* button creation routine. This will cause a small line to be added above the *Quit* button in the pull-down menu, causing a visual separation between the buttons. This is a small thing, but it is enormously important for writing graphical user interfaces that are intuitive for users. The *Save As* menu should look similar to the illustration in Figure 114.

Notice too that we have placed a file extension in the user value of each button. We can use this file extension to construct a default file name in the event handler.

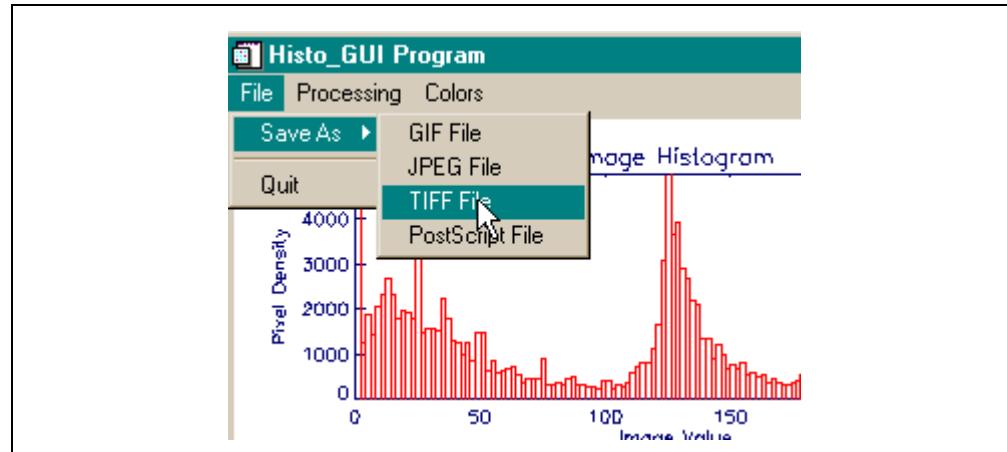


Figure 114: The Save As menu in the *Histo_GUI* program. Note the separator line above the Quit button. This provides the user with a visual clue that the Quit operation is very different from a Save As operation.

Writing the File Output Event Handler

The next step is to write the *Histo_GUI_File_Output* event handler. As before, the skeleton of the event handler can be written immediately. It is written like this and added to the program file somewhere before the *Histo_GUI* widget definition module.

```
PRO Histo_GUI_File_Output, event
Widget_Control, event.top, Get_UValue=info, /No_Copy
Widget_Control, event.top, Set_UValue=info, /No_Copy
END
```

Normally any additional event handler code will appear between the lines that get and set the *info* structure into the user value of the top-level base. However, in file output event handlers we sometimes make an exception to this rule. The reason for this is that we typically need to ask the user for some information. For example, we may want the user to select a name for the output file.

The modal dialog widget that we often use to gather this information will almost always have a *Cancel* button. If the user selects the *Cancel* button, then the program should exit the event handler without doing any damage. If the modal dialog does not require any information from the *info* structure, it is often easier to check for the cancel operation before the *info* structure is checked out. That will be the case here.

For the GIF, JPEG, and TIFF files we are going to make some arbitrary decisions about how to write the files. The only piece of information we need from the user is the name of the output file. We can obtain this using the *Dialog_Pickfile* command with the *Write* keyword set. For PostScript files, we are going to allow the user more input into how the file should be configured. That input is going to be collected with the *PSCConfig* program you downloaded to use with this book. (See “Configuring the PostScript Device with PSCConfig” on page 198.)

The first piece of information we might want to collect is the depth of the visual display and the color decomposition state, because you recall that this information is critical when writing GIF, JPEG, and TIFF files. (See “Reading and Writing Files with Popular File Formats” on page 147.) Type this command (in bold), just after the procedure definition statement:

```
PRO Histo_GUI_File_Output, event
Device, Get_Visual_Depth=theDepth, Get_Decomposed=theState
```

```
Widget_Control, event.top, Get_UValue=info, /No_Copy
```

Next, we can find out which button was selected and gather the file extension we want to use to construct the initial default filename. Recall that we stored the appropriate file extension in the user value of each button. Add these two commands (in bold) directly after the command you just typed:

```
Widget_Control, event.id, Get_Value=buttonValue, $
Get_UValue=file_extension
startFilename = 'histo_gui' + file_extension

Widget_Control, event.top, Get_UValue=info, /No_Copy
```

Now we can collect information from the user. If the user wants a PostScript file, we can gather the appropriate *Device* keywords to configure the PostScript device with the *PSConfig* program. Otherwise, we can obtain the output file name with the *Dialog_Pickfile* command. Type these commands (in bold) just below the commands you just typed:

```
IF buttonValue EQ 'PostScript File' THEN BEGIN
    keywords = PSConfig(Cancel=cancelled, $
        Filename=startFilename, Group_Leader=event.top)
    IF cancelled THEN RETURN
ENDIF ELSE BEGIN
    filename = Dialog_Pickfile(File=startFilename, /Write)
    IF filename EQ "" THEN RETURN
ENDIFELSE

Widget_Control, event.top, Get_UValue=info, /No_Copy
```

Notice that you can use the *Cancel* keyword to determine if the user cancelled out of the *PSConfig* program, but that *Dialog_Pickfile* works differently. In *Dialog_Pickfile* you must check to see if the return value is a null string. Note also that you must use the *Group_Leader* keyword with *PSConfig* if you expect modal widget behavior.



European users of *PSConfig* might want to set the *European* keyword in the call. This will select an A4 paper size as the default paper size, rather than the 8.5 x 11 inch American paper size.

All of the code so far has gone in front of the code line that gets the *info* structure out of the user value of the top-level base. But if we get through this part of the code, we are ready for the *info* structure.

To create GIF, JPEG, and TIFF files, we have to take a snapshot of the display window. We better be sure we know which window that is, or we will be taking a snapshot of a window in some other program. Remember, there can only be one current graphics window in IDL at any one time. The next line of code (in bold) will make sure the current graphics window is the display window of this program.

```
Widget_Control, event.top, Get_UValue=info, /No_Copy

WSet, info.wid
```

Finally, we come to the *CASE* statement where the code to produce the different file types will be located. Let's write the skeleton *CASE* statement and then come back and write the file output code one by one. The skeleton *CASE* statement (in bold) looks like this:

```
CASE buttonValue OF
'GIF File': BEGIN
END
'JPEG File': BEGIN
END
```

```

'TIFF File': BEGIN
    END
'PostScript File': BEGIN
    END
ENDCASE
Widget_Control, event.top, Set_UValue=info, /No_Copy
END

```

Creating the GIF File

Creating a color GIF file with *Write_GIF* requires that you have a file name, a 2D image array, and the red, green, and blue vectors that describe a color table for the image. (Note that starting with IDL 5.4 you must obtain a license from the GIF file patent holder to create GIF files.) You recall that how you obtain a 2D image array and the color vectors depends upon the display depth of your machine. (See “Creating Color GIF Files” on page 150.)

If you are on an 8-bit machine, you can get the 2D image directly from a snapshot of the display and the color vectors from the current color table. If you are on a 24-bit machine, you have to convert the snapshot of the display to a 2D array and the color vectors with *Color_Quan*. You must also be sure you have set color decomposition on when you take the snapshot, or will we get incorrect colors on PCs and Macintosh computers.

The code (in bold) can be entered into the GIF file portion of the *CASE* statement above. It looks like this:

```

'GIF File': BEGIN
    IF theDepth GT 8 THEN BEGIN
        Device, Decomposed=1
        snapshot = TVRD(True=1)
        Device, Decomposed=theState
        image2D = Color_Quan(snapshot, 1, r, g, b, $
            Colors=256, /Dither)
    ENDIF ELSE BEGIN
        TVLCT, r, g, b, /Get
        image2D = TVRD()
    ENDELSE
    Write_GIF, filename, image2D, r, g, b
END

```

Note that the *Colors* keyword for the GIF file has been set to 256 and the *Dither* keyword has been turned on. These are details that you may want to change for your own purpose, or perhaps you want them to be set by the user and would like to write some kind of dialog form widget (as described in the next chapter) that could collect the user’s instructions for you.

Creating the JPEG File

Creating the color JPEG file also depends on the depth of the visual display, but in just the opposite way that the GIF file did. (See “Creating Color JPEG Files” on page 154.) To write a JPEG file with *Write_JPEG*, you need a file name and a 24-bit image. You can create a 24-bit image directly from a screen snapshot on a 24-bit display, while on an 8-bit display you have to construct a 24-bit image from a 2D snapshot and the color vectors.

The code (in bold) can be entered into the JPEG file portion of the *CASE* statement above. It looks like this:

```
'JPEG File': BEGIN
    IF theDepth GT 8 THEN BEGIN
        Device, Decomposed=1
        image3D = TVRD(True=1)
        Device, Decomposed=theState
    ENDIF ELSE BEGIN
        image2D = TVRD()
        TVLCT, r, g, b, /Get
        s = Size(image2D, /Dimensions)
        image3D = BytArr(3, s[0], s[1])
        image3D[0,*,*] = r[image2d]
        image3D[1,*,*] = g[image2d]
        image3D[2,*,*] = b[image2d]
    ENDELSE
    Write_JPEG, filename, image3D, True=1, Quality=85
END
```

Note that the *Quality* keyword is set to 85. This is a fairly high quality image, without too much compression. (The default value is 75.) Again, you may want the user to have some input on this, but here we are selecting the quality arbitrarily.

Creating the TIFF File

Creating a color TIFF file is almost identical to creating a color JPEG file. (See “Creating Color TIFF Files” on page 156.) The *Write_TIFF* command requires a file name, and a 24-bit image. The only trick with TIFF images is that you want to flip them upside-down with the *Reverse* command before you write them to a file.

The code (in bold) can be entered into the TIFF file portion of the *CASE* statement above. It looks like this:

```
'TIFF File': BEGIN
    IF theDepth GT 8 THEN BEGIN
        Device, Decomposed=1
        image3D = TVRD(True=1)
        Device, Decomposed=theState
    ENDIF ELSE BEGIN
        image2D = TVRD()
        TVLCT, r, g, b, /Get
        s = Size(image2D, /Dimensions)
        image3D = BytArr(3, s[0], s[1])
        image3D[0,*,*] = r[image2d]
        image3D[1,*,*] = g[image2d]
        image3D[2,*,*] = b[image2d]
    ENDELSE
    Write_TIFF, filename, Reverse(Temporary(image3D),3)
END
```

The *Temporary* command causes the memory occupied by the *image3D* array to be used *in situ*, thereby saving memory allocation.

An Alternative Way of Creating GIF, JPEG, and TIFF Files

There is an alternative way to create GIF, JPEG, and TIFF files. (As well as BMP, PNG, and PICT files.) The *TVRead* command you downloaded to use with this book simplifies taking a screen snapshot and writing an output file tremendously. In fact, you could, if you wanted to, re-write the GIF, JPEG, and TIFF code above like this:

```
CASE buttonValue OF
```

```

'GIF File' : image = TVRead(Filename=filename, /GIF)
'JPEG File': image = TVRead(Filename=filename, /JPEG)
'TIFF File': image = TVRead(Filename=filename, /TIFF)
'PostScript File': BEGIN
    END
ENDCASE

```

The program intelligence incorporated into this code you just wrote has been built into the *TVRead* command, just as much of the required image display intelligence has been built into the *TVImage* command.

Creating the PostScript File

Creating the PostScript file will not be difficult because of the way we wrote the graphics display portion of the code. Recall we went to considerable effort to not include commands like *Window* and *WSet* (see “The *HistoImage* Program is Device Independent” on page 255 for additional information) in the *HistoImage* program, which are not appropriate commands for the PostScript device. But PostScript wouldn’t be PostScript if there wasn’t something tricky about it. (See “Printing PostScript Files” on page 180.) Let’s see what some of the problems might be.

To create a PostScript file you have to have some way of recreating the commands that created the display window in the first place. We have such a way in the *HistoImage* program. The only other thing that can cause major problems in PostScript files is color. And here is where we have to be careful.

Recall that I said you can have any background color you like in a PostScript file as long as that color is white. This is a facetious way of pointing out that there is a difference between the way graphics are drawn on the display and the way they are written to a file. (See “Problem: PostScript Devices Use Background and Plotting Colors Differently” on page 189 for more information.) In particular, filling the background color is a raster operation, while most graphics commands of the type written into a PostScript file are vector operations.

So, right away we have a good chance that what the user sees on the display is not going to look the same in a PostScript file.

Then we have the problem that if the user was looking at a display with a dark background color, they were almost certainly using light annotation and drawing colors for contrast. These colors will certainly not show up well on a white background. So these drawing colors might need to be changed before we write the PostScript output.

Suppose we decide to solve the drawing color problems by not using any drawing colors at all. This would cause *HistoImage* to fall back on its default drawing colors, which we happened to choose for their compatibility with PostScript output. Perhaps not perfect, but a workable compromise. We will use it.

Which brings us to the problem of image colors. The original *HistoImage* program loaded its own color table, which is perfectly compatible with PostScript. But in the *Histo_GUI* program, we decided to control the image colors outside of the *HistoImage* program. This will cause us a problem if we are running the program on a display that does not have as many colors as the PostScript device’s 256. In other words, we will have a problem running the program on an 8-bit display.

Consider this situation. Suppose IDL is running on a Windows computer with an 8-bit graphics card. IDL will use 236 colors, since Windows reserves 20. (This situation is no different from running on a UNIX system with an 8-bit display.) In this case *!D.Table_Size* is equal to 236. Our drawing colors are loaded into color indices 233,

234, and 235. Our image colors are loaded in indices 0 to 232. We have loaded the colors and saved the color vectors in the *info* structure.

Now we make the PostScript device the current graphics and we run the *HistoImage* program to draw graphics. Because there are 256 colors in the PostScript device, the drawing colors are loaded in indices 252, 253, and 254. The image colors should be loaded in indices 0 to 251, which are the indices that the image data is scaled into. But we are not loading the color table. Because if we do there is an excellent chance that what appears in the PostScript file will not look anything at all like what the user saw on the display. (For example, the user might have stretched the color table, or changed the gamma function.) If we don't load a color table, we can at least copy the color vectors into the PostScript file. But those color vectors use indices 0 to 235, and they contain our drawing colors!

Our output will look similar to the illustration in Figure 115. Notice the colors from

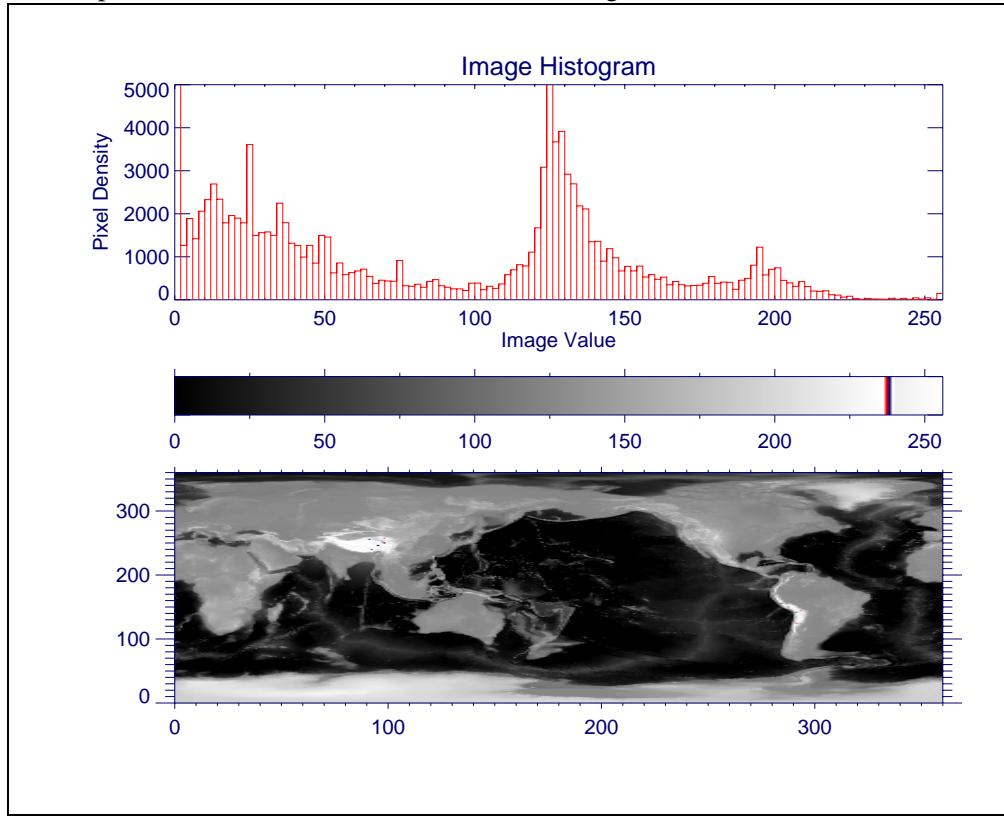


Figure 115: The PostScript file created on a machine with an 8-bit display before the image colors were re-sampled into the number of image colors available in the PostScript device.

index 232 on are not correct. And you can see vividly the previous drawing colors (the bands across the color bar). One possible solution is to rebin the image colors from the color vectors stored in the *info* structure into the number of image colors used in the PostScript device. In other words, re-sample the 232 colors from the 8-bit display into 252 colors. The code to do the re-sampling and load the color table in the PostScript device (don't add these commands to your file yet) will use the *TVLCT* and *Congrid* commands, and look like this:

```
topcolor = info.imagecolors-1
TVLCT, Congrid(info.r[0:topcolor], !D.Table_Size-4), $
Congrid(info.g[0:topcolor], !D.Table_Size-4), $
```

```
Congrid(info.b[0:topcolor], !D.Table_Size-4)
```

Another point to consider in creating PostScript output is the kind of fonts you would like to use. Hardware or true-type fonts almost always look better in PostScript output than the vector fonts typically used on the display. Changing the font makes the output look slightly different from the output on the display. (See “Problem: PostScript Devices Can Use Different Display Fonts” on page 187 for more information.) But it is essential if you want nice looking PostScript output. Be sure to use true-type fonts if you have text rotated in 3D space. If your text is only rotated in 2D space, as here, then hardware fonts are perfectly acceptable.

So, given these considerations, the complete text of the PostScript portion of the *CASE* statement (in bold) looks like this:

```
'PostScript File': BEGIN
    thisDevice = !D.Name
    thisFont = !P.Font
    !P.Font = 0
    Set_Plot, 'PS'
    IF theDepth EQ 8 THEN BEGIN
        topColor = info.imagecolors-1
        ncolors = !D.Table_Size-4
        TVLCT, Congrid(info.r[0:topColor], ncolors), $
            Congrid(info.g[0:topColor], ncolors), $
            Congrid(info.b[0:topColor], ncolors)
    ENDIF
    Device, _Extra=keywords
    HistoImage, *info.process, $
        Binsize=info.binsize, $
        _Extra=*info.extra, $
        Max_Value=info.max_value, $
        NoLoadCT=1, $
        XScale=info.xscale, $
        YScale=info.yscale
    Device, /Close_File
    Set_Plot, thisDevice
    !P.Font = thisFont
END
```

Notice that the *!P.Font* system variable is set to use hardware fonts (*!P.Font=0*). The image colors are scaled into the correct number of Postscript colors if the program is running on 8-bit displays. And that the *AxisColorName*, *BackColorName*, and *DataColorName* keywords have been left off the call to *HistoImage*, thereby invoking the default drawing colors for the program.

Notice too the *Device* command with the *Close_File* keyword set. This command is essential if you want to be able to print the resulting PostScript file.

The PostScript output will look similar to the illustration in Figure 116.

Save your program, re-compile it, and run it. (If you did not enter the code above into a file named *histo_gui.pro*, you can use the file *histo_gui.5.pro*, which you downloaded to use with this book. This file must be explicitly compiled to be used. See “The Advantage of Mistakes” on page 261 for more information.)

```
IDL> .Compile histo_gui
IDL> Histo_GUI
```

Does the program work the way you expect it to? Good. Then let’s add a *Print* capability.

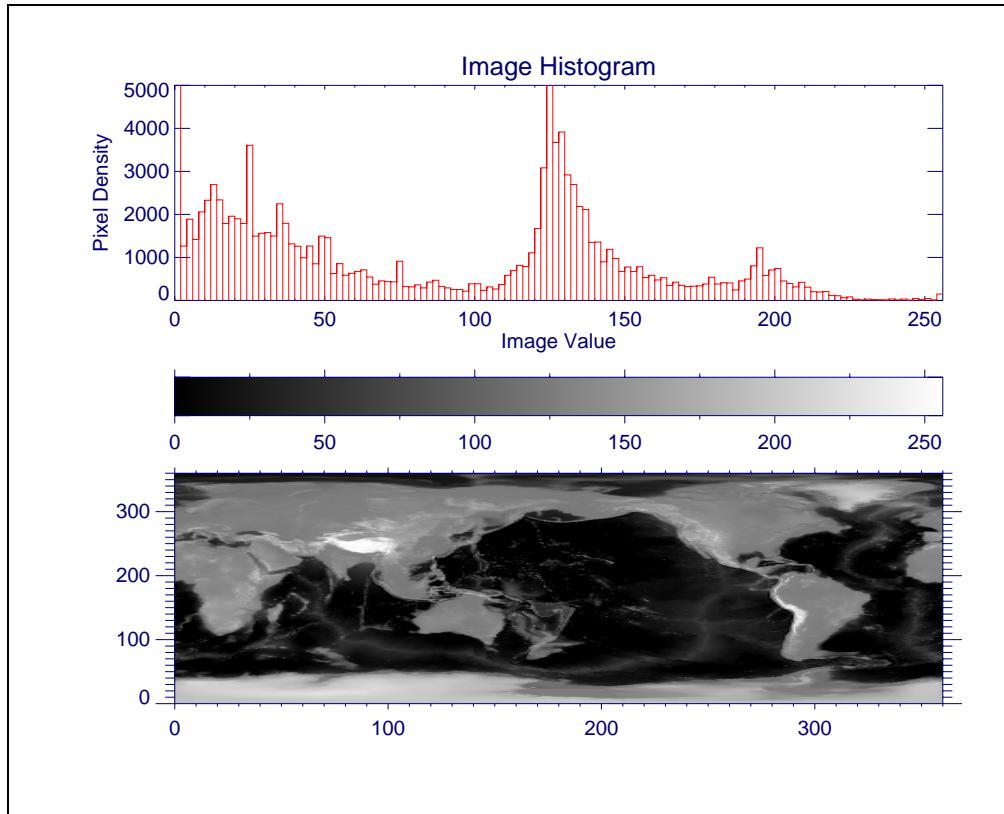


Figure 116: The PostScript file output from the *Histo_GUI* program.

Adding Printer Functionality

The next piece of functionality we might like to add to our program is the ability to send the graphics display directly to a printer. This is fundamentally like writing a PostScript file, except that the output device is the *Printer* device rather than the PostScript device. This is especially so with respect to color output.

Creating the Print Pull-Down Menu

Let's add the *Print* capability to the program as a pull-down menu item under the *File* menu. We are going to have the option of printing in portrait or landscape mode. Add the following code (in bold) to the *Histo_GUI* widget definition module. The *Print* button will be the first option under the *File* button.

```
fileID = Widget_Button(menuID, Value='File')
printID = Widget_Button(fileID, Value='Print', $
    Event_Proc='Histo_GUI_Print', /Menu)
button = Widget_Button(printID, Value='Portrait Mode')
button = Widget_Button(printID, Value='Landscape Mode')
```

Notice that the events from the *Portrait* and *Landscape* mode buttons will “bubble-up” to the event handler associated with the *Print* button.

Writing the Print Event Handler

As before, we can immediately write the *Histo_GUI_Print* skeleton event handler. Add it somewhere in the file in front of the *Histo_GUI* widget definition module.

```

PRO Histo_GUI_Print, event
Widget_Control, event.top, Get_UValue=info, /No_Copy
Widget_Control, event.top, Set_UValue=info, /No_Copy
END

```

Recall that we asked the user to provide information in the file output event handler before we got the *info* structure out of the user base of the top-level base. We did that because if the user cancels out of the dialog, we can return without having to worry about checking the *info* structure back in. We want to do the same thing here by calling the *Dialog_PrinterSetup* dialog. This will give the user a chance to select the default printer and configure the printer the way they like. For example, the user could select the number of copies to print or could even elect to create a file from the output instead of sending it to the printer. (Not all printer dialogs allow you to write to a file. It depends on your printer.)

So, at the first line in the program module after the procedure definition statement, type the two lines (in bold) below.

```

PRO Histo_GUI_Print, event
ok = Dialog_PrinterSetup()
IF NOT ok THEN RETURN
Widget_Control, event.top, Get_UValue=info, /No_Copy

```

The next step is to get ready to print. What you do here depends somewhat on your printer. I like to use true-type fonts (*P.Font=1*) and I typically increase the thickness of line plots and annotation slightly. I do this because on my 600 dpi printer, a single pixel line can be too light to reproduce properly. We also have to rescale the color table vectors if the current device has fewer colors than the *Printer* device. (We just encountered this problem in creating PostScript file output.) And I like to set the hourglass cursor, to alert the user that printing can take a couple of seconds to accomplish. Type the following code (in bold) just after the line that gets the *info* structure:

```

Widget_Control, event.top, Get_UValue=info, /No_Copy
thisDevice = !D.Name
Device, Get_Visual_Depth=theDepth
thisFont = !P.Font
thickness = !P.Thick
!P.Font=1
!P.Thick = 2
Widget_Control, /Hourglass

```

Note that the hourglass cursor will return to the normal cursor when we exit this event handler.

The next step is to configure the *Printer* device, based on whether the user wants portrait or landscape output. I've noticed on some printers (not all) that I cannot select the page mode at the same time that I am selecting the size of the output and the offsets on the page. I don't know why. But I've learned to set the page mode with a *Device* command of its own, and then set the other printer properties with another *Device* command.

I use the *PSWindow* command to position the output on the printer page. (The *PSWindow* program is among the programs you downloaded to use with this book. It is discussed in “Positioning Graphics with the Printer Device” on page 202.) Recall that the *Printer* keyword must be set for this command, otherwise the landscape offsets will be incorrect. And, recall, that for some printers, the offsets are calculated from the printable edge of the page, rather from the actual corner of the page. The *Fudge* keyword accounts for an additional 0.25 inch offset that has to be subtracted

from the *PSWindow*-calculated offsets to produce centered output on my printer. You may have to experiment with these values for your printer. European users might also want to set the *Pagesize='A4'* keyword to calculate sizes and offsets based on an A4 paper size.

The following code should be added to the program immediately after the lines you just typed above:

```
Widget_Control, event.id, Get_Value=buttonValue
CASE buttonValue OF
    'Portrait Mode': BEGIN
        keywords = PSWindow(/Printer, Fudge=0.25)
        Set_Plot, 'PRINTER', /Copy
        Device, Portrait=1
    ENDCASE
    'Landscape Mode': BEGIN
        keywords = PSWindow(/Printer, /Landscape, Fudge=0.25)
        Set_Plot, 'PRINTER', /Copy
        Device, Landscape=1
    ENDCASE
ENDCASE
Device, _Extra=keywords
```

Notice the button value is used in the *CASE* statement to branch to the appropriate code, and that the keyword inheritance mechanism is being used to pass the *PSWindow* keywords to the *Device* command.

The next step is to stretch the color vectors if the program is running on an 8-bit display. Type these commands immediately following the commands above:

```
IF theDepth EQ 8 THEN BEGIN
    topColor = info.imagecolors-1
    TVLCT, Congrid(info.r[0:topColor], !D.Table_Size-4), $
    Congrid(info.g[0:topColor], !D.Table_Size-4), $
    Congrid(info.b[0:topColor], !D.Table_Size-4)
ENDIF
```

Next, we are ready to display the graphics. Notice that I am going to set the data and annotation colors to black, because I am outputting this to a gray-scale printer. You may wish to use the current colors on the display, or to change the output to some other color if you are outputting to a color printer (e.g., an HP color Inkjet printer). Notice that I am not setting the background color, however, as that color is going to be white no matter what I set the value to. (The *Printer*, like the PostScript, device does not draw the background color.)

Type this command immediately following the commands above in the file:

```
HistoImage, *info.process, $
    AxisColorName='Black', $
    Binsize=info.binsize, $
    DataColorName='Black', $
    _Extra=*info.extra, $
    Max_Value=info.max_value, $
    NoLoadCT=1, $
    XScale=info.xscale, $
    YScale=info.yscale
```

Finally, we close the *Printer* device, and clean-up. Type these commands (in bold) just before the command that puts the *info* structure back in the user value of the top-level base.

```
Device, /Close_Document
Set_Plot, thisDevice
!P.Font = thisFont
!P.Thick = thickness

Widget_Control, event.top, Set_UValue=info, /No_Copy
END
```

Save the file, re-compile it, and test it. (If you did not enter the program above into a file named *histo_gui.pro*, you can use the file *histo_gui.6.pro*, which you downloaded to use with this book. This file must be explicitly compiled to be used. See “The Advantage of Mistakes” on page 261 for more information.)

```
IDL> .Compile histo_gui
IDL> Histo_GUI
```

Can you get output from your printer? Good. Then you are ready to move onto the next chapter, where you will learn even more widget programming techniques.

Chapter 12

◆ Discovering the Possibilities ◆◆◆



Creating Dialog Form Widgets

Chapter Overview

The purpose of this chapter is to demonstrate two ways to create *dialog form widgets*. Dialog form widgets are widget programs that collect information from the user and pass that information along to another widget program module or to another widget program altogether. Among the techniques you will learn are these:

- How to write modal or blocking dialog form widgets
- How to write non-blocking dialog form widgets
- How to use pointers to store information in widget programs
- How to use pseudo events in widget programs
- How to pass information between independent widget programs

Creating a Modal Dialog Form Widget

If you wrote the *Histo_GUI* program in the last chapter you have already used a modal widget dialog program: the *PickColorName* program allowed the user to select a different drawing color. (If you didn't write the *Histo_GUI* program in the last chapter, you can use the file *histo_gui.6.pro* that you downloaded to use with this book. Rename the file as *histo_gui.pro*. Compile and run the program like this:

```
IDL> .Compile histo_gui  
IDL> Histo_GUI
```

Try changing one of the drawing colors in the program by selecting *Drawing Colors* item under the *Colors* menu. Notice that while the *PickColorName* program is on the display that you cannot work with the *Histo_GUI* program. You must dismiss the *PickColorName* program by selecting either the *Cancel* or the *Accept* button to use the *Histo_GUI* program again. This is what is meant by a *modal* widget program.

Another way to use the program is from the IDL command line. For example, you can type this:

```
IDL> color = PickColorName() & Print, color
```

When the program is called in this way you will notice that the IDL command line prompt is dimmed, indicating that IDL is not responding to command line input. You must dismiss the *PickColorName* program by selecting either the *Cancel* or *Accept*

button to release the block on the IDL command line. In this case, the program is said to be a *blocking* widget program.



There are several subtle differences between a modal and a blocking widget. If you want to write dialog form widgets, you must understand the difference between a blocking and a modal widget and how they work, exactly.

A Blocking Widget Program

When called from the command line, the *PickColorName* program runs as a *blocking* widget program.:.

```
IDL> color = PickColorName()
```

Notice that your IDL command line prompt either disappears or gets dim. It is impossible to enter any more commands at the IDL command line and have them executed. We say that the command line is *blocked*. (It is possible to enter commands when you have a dimmed command line prompt, but they are not executed until the command line is unblocked.)

Normally, any IDL program you run blocks the IDL command line. In other words, you can't type another command and have it executed until the command currently executing is finished. Up until IDL 5, the same could be said about widget programs. Once you started a widget program running, you had no access to the IDL command line until that widget program finished executing.

But this behavior changed in IDL 5 and it is now possible to have a widget program running *and* have access to the IDL command line where you can type commands and have them executed immediately. We call such a widget program a *non-blocking* widget program. To create a non-blocking widget program, you use the *No_Block* keyword on the program's *XManager* command. This is exactly what you did with the *Histo_GUI* program you wrote earlier.

The *XManager* command in *PickColorName*, however, is written without the *No_Block* keyword. Hence, it is a blocking widget.

What “blocking” means in this context is that IDL stops executing the code in the *PickColorName* widget definition module as soon as the *XManager* command is executed. Any code below the *XManager* command in the widget definition module (and there is some, as you will see in a moment) is not executed until the widget program is destroyed, thus freeing the block.

This is perfect for a program like *PickColorName* because the block gives the user time to fill out the information on the form or dialog. When the user gets the form filled out, they click either the *Cancel* or *Accept* button. In either case, the widget is destroyed, releasing the block and the program is free to do whatever it likes with the information it has collected. In the case of *PickColorName*, this means collecting the name of the selected color and then returning that name to the user as the result of the function. All of this collecting and returning is done *after* the *XManager* command in the widget definition module of the program.



All good so far. But here is the subtle part of blocking that is easy to overlook. It is only the *first* program that calls *XManager* as a blocking widget program that blocks! All subsequent blocking programs run *through* their blocks and *act* as if they were non-blocking widget programs.

This kind of behavior can be a complete disaster for a program like *PickColorName*, because the program will return a color name even before the user has had a chance to touch the form! As some of you may imagine, that will make it hard to change colors in your program.

A Modal Widget Program

If you want to make sure that a widget program blocks under all circumstances, and not just if it is the lucky first one to make a blocking *XManager* call, then you must make that widget program a *modal* widget program. A modal widget program *always* blocks at the *XManager* command until the widget is destroyed.

Prior to IDL 5, it was relatively simple to create a modal widget program. You simply set the *Modal* keyword on the *XManager* command. But in IDL 5 the *Modal* keyword on the *XManager* command was made obsolete. It has been replaced by a *Modal* keyword on the *Widget_Base* command that creates the top-level base for the program.

There is just one small, but important, complication. If you set the *Modal* keyword for a base widget, you must also specify a valid group leader (via the *Group_Leader* keyword) for that base widget. This makes it just a little more difficult to write a dialog widget that works both at the IDL command line and in an IDL widget program. But you will see what I mean as you write the following program.

Writing a Modal Dialog Form Widget Definition Module

The *PickColorName* program is a good example of a modal dialog form widget. It is a function that returns a value to the *Histo_GUI* program that is required to continue program execution. But the *Histo_GUI* program needs a method of loading a new image once the program is running. I think it would be a good idea to stop now and build a simple modal dialog form widget to collect file information about an image file, so the *Histo_GUI* program can open the file and read the image. The information you need to collect about the image file will be the name of the file and the X and Y size of the image data in the file. (To keep the program relatively simple, we will assume the image file doesn't have a header, that the image is 2D, and that the data is byte data. An assumption that will be true for many, but not all, of the images in the IDL distribution.) Let's name the dialog form widget we wish to create *OpenImage*.



You can learn more about some of the data files in the IDL distribution, including their file sizes, by referring to “Appendix B: Data File Descriptions” on page 397. These files are normally found in the *examples/data* subdirectory of the main IDL distribution. If you prefer to copy these files to another directory, change to that directory and then type *CopyData* at the IDL command prompt. A dialog will ask you if you are in the proper subdirectory. Selecting the *OK* button will permit IDL to copy all of the needed data files to your current directory.

```
IDL> CD, './coyote'
IDL> CopyData
```

You are going to be asking the user to type some information into a form. In general, this is not a good idea because—if we know anything at all about users—we know they can't type. It would be a better idea to have the user pick a file name or select a file size with the mouse, if possible. But I suppose it does us all some good to live recklessly once in a while. We will make a slight compromise and give the user the ability to select a file name with the mouse, since file names are much harder to type correctly than a file size.

Given that you are going to ask the user to type, it would be a good idea to have him or her type as little as possible. One way to do that is to provide some good default values in the typing fields. Perhaps the value will be the correct one, and the user won't *have* to type. You would also like the user to be able to specify a particular file name or file size when the program is called. So, open a text editor and define the function definition statement of your dialog form widget like this:

```
Function OpenImage, $
```

```

        Filename=filename, $           ; Initial filename.
        Group_Leader=group_leader, $ ; Group leader identifier.
        XSize=xsize, $              ; Initial X size.
        YSize=ysize, $              ; Initial Y size.
        Cancel=cancel                ; A cancel flag. (Output)

```

The *Cancel* keyword will be an output variable that will indicate whether it was the *Cancel* or *Accept* button that was clicked on the dialog. The *Group_Leader* keyword will hold the identifier of the group leader for this modal widget program. Remember that a group leader is required to properly define a modal base widget.

Next, decide what you are going to do if an error occurs in this program module (I like to return to the caller of the program if the program module is a function) and provide default values for the keywords if they are not present. Add this:

```

On_Error, 2 ; Return to caller.
IF N_Elements(filename) EQ 0 THEN $
    filename=Filepath('ctscan.dat', $
                      SubDirectory=['examples','data'])
IF N_Elements(xsize) EQ 0 THEN xsize = 256
IF N_Elements(ysize) EQ 0 THEN ysize = 256

```

Here the *Filepath* command is used to construct a filename to the *ctscan.dat* image file in the *examples/data* subdirectory of the main IDL directory. (The main IDL directory is the one pointed to by the *!Dir* system variable.) The *Filepath* command returns a device-appropriate file name. The default file will be *ctscan.dat*, which is a 256 by 256 byte array.

Defining a Modal Top-Level Base

The next step is to create the top-level base widget for this modal program. I've already mentioned several times that a valid group leader is required to define a modal top-level base widget. That is not a problem if a group leader is passed to the program via the *Group_Leader* keyword. But this will not always be the case. For example, if the user wanted to call this program from the IDL command line it would not be possible (in general) to define a valid group leader. Without a valid group leader, you will have to rely on the program being a blocking widget. Again, this is not a problem if the program is called from the IDL command line, since—by definition—if you are at the command line this program will be the first blocking program to call *XManager*.

Where you *can* get into trouble is if the group leader is not defined and the program from which the *OpenImage* program is called is *itself* a blocking widget program. I don't know any way around this dilemma. And what I dislike even more is that it makes the group leader a required parameter if this program is to be called correctly from a widget program. And yet, it can't be a required parameter if the program is to be used at the IDL command line. Life is not always simple, I guess.



If you examine widget code that is supplied with IDL in the *lib* sub-directory, you often find Research Systems programmers trying to work around the absence of a group leader in creating modal widgets by using an unrealized base widget as the group leader. For example, you will see code like this:

```

IF N_Elements(group_leader) EQ 0 THEN $
    group_leader = Widget_Base()

```

This is a completely unsatisfactory solution, in my opinion, since on Windows machines (and possibly others) there is a requirement that the group leader cannot be an unrealised widget. As a result this “unrealized” base widget gets realized when the program that now belongs to its group is realized. This results in a tiny widget window in the upper left-hand corner of the display that the user can neither grab with the

mouse nor remove. Windows on the display that I can't control annoy me more than I can safely say in a book!

Anyway, if the group leader is specified, make the top-level base modal. If the group leader is not specified, then you will have to hope the program is being called from the IDL command line.

```
IF N_Elements(group_leader) NE 0 THEN $
    tlb = Widget_Base(Column=1, $
                      Title='Enter File Information...', $
                      Group_Leader=group_leader, /Modal, /Floating, $
                      /Base_Align_Center) ELSE $
    tlb = Widget_Base(Column=1, $
                      Title='Enter File Information...', $
                      /Base_Align_Center)
```

Notice the *Floating* keyword set on the modal top-level base. A *floating* widget always floats above the program that is its group leader. This prevents the program from getting lost behind other windows. The *Base_Align_Center* keyword makes sure that the children of this top-level base are centered in the top-level base.

Defining Other Widgets

Next, define two sub-bases to organize the interface. Notice that these bases are also column and row bases. Organizing the widget layout with bases of this type (rather than bases that are explicitly sized) results in widget programs that are platform independent. The base widgets size themselves to fit their contents. Type this:

```
subbase = Widget_Base(tlb, Column=1, Frame=1)
filebase = Widget_Base(subbase, Row=1)
```

Notice the *Frame* keyword in the first *Widget_Base* command. This will put a frame around this base one pixel wide, giving a visual clue to the user that the items in this base are organized together.

Next, create the widgets that go into the base widgets. Type:

```
filesize = StrLen(filename) * 1.25
fileID = CW_Field(filebase, Title='Filename:', $
                  Value=filename, XSize=filesize)
browseID = Widget_Button(filebase, Value='Browse', $
                         Event_Pro='OpenImage_BrowseFiles'
                         xsizeID = CW_Field(subbase, Title='X Size:', $
                                           Value=xsize, /Integer)
                         ysizeID = CW_Field(subbase, Title='Y Size:', $
                                           Value=ysize, /Integer)
```

Notice that the compound widget *CW_Field* is used for the text fields. The purpose of *CW_Field* is to put a label widget beside an editable text widget. But the *CW_Field* event handler can take care of a lot of the details for you. For example, by telling *CW_Field* that the value in the size widgets should be an integer it will make sure the user can *only* enter integers in this field. Plus, when you ask for the value of this text widget it will be returned to you as an integer, not as a text array, which it would be otherwise. This makes working with these text widgets much easier.

The *Browse* button also has its own event handler, named *OpenImage_BrowseFiles*, which is assigned with the *Event_Pro* keyword.

Next, create a base to hold the *Cancel* and *Accept* buttons and those two buttons. Type:

```
butbase = Widget_Base(tlb, Row=1)
```

```
cancel = Widget_Button(butbase, Value='Cancel')
accept = Widget_Button(butbase, Value='Accept')
```

I like my modal dialog widgets to come up in the center of the display, where the user *must* see them. (You can imagine why I dislike using *Dialog_Pickfile*, an IDL-supplied modal dialog widget which always comes up the upper right-hand corner of the display. Sigh...) So I would like to center this dialog on the display. But I want to do it *before* the widget hierarchy is realized.

At this point, all the widgets have been created, they just haven't been realized. But I can use the *Widget_Info* command with the *Geometry* keyword set to get geometric information about the top-level base widget. In particular, I would like to know the size of the top-level base widget in device or pixel units. This information will be in the *Scr_XSize* and *Scr_YSize* fields of the structure returned by the *Widget_Info* command. Using this information, along with the size of the display, will allow me to calculate the X and Y offsets necessary to center the top-level base on the display. The code will look like this:

```
screenSize = Get_Screen_Size()
geom = Widget_Info(tlb, /Geometry)
Widget_Control, tlb, $
XOffset = (screenSize[0] / 2) - (geom.Scr_XSize / 2), $
YOffset = (screenSize[1] / 2) - (geom.Scr_YSize / 2)
```

Since I center many widgets, I have wrapped the code above in a small program named *CenterTLB*, that you downloaded to use with this book. If that file is in the IDL path, then I can replace the code above with a single line of code:

```
CenterTLB, tlb
```

At this point the widgets have been created and centered, so to realize the program, you must type this line of code:

```
Widget_Control, tlb, /Realize
```

Storing Collected Information in Modal Dialogs

The idea of a dialog form widget is to collect information from the user and then return it to the user (or do something with the information) when the user clicks the *Accept* button. But when the user clicks the *Accept* button the widget program is destroyed, releasing the modal blocking action. The question to be asked then, is this: Where should the information collected by the program be stored so it can be processed or returned to the user?

It clearly won't do to store the information somewhere inside the program (say, in a user value) because it will be destroyed when the program is destroyed. So it has to be stored somewhere *external* to the program. A common block is one answer. But since I never use common blocks in widget programs, I prefer to store form information in pointers. Create a pointer for this program like this:

```
ptr = Ptr_New({Filename: '', Cancel:1, XSize:0, YSize:0})
```

This pointer will point to an anonymous structure containing fields for all of the information you wish to collect from the form. Notice that there is a field named *Cancel* to let you know if the user clicked the *Cancel* button or the *Accept* button. This is important information, because you will want to do different things depending upon this action.

I typically set the information in the pointer to look as if the user canceled out of the program. I do this so that if an error occurs in the program, I can simply return the starting pointer structure. If the user has taken advantage of the *Cancel* keyword (as

they are supposed to) and is checking it to see whether to continue program execution, then errors will not interrupt program operation.

Creating the Info Structure

Like any other widget program, this one needs an *info* structure with all of the information required to run the program. In this case, you need the identifiers of the widgets holding information you want to collect from the user and the location where that information should be stored. The *info* structure will be defined and stored in the user value of the top-level base like this:

```
info = {fileID:fileID, xsizeID:xsizeID, $  
        ysizeID:ysizeID, ptr:ptr}  
Widget_Control, tlb, Set_UValue=info, /No_Copy
```

Creating a Blocked Widget

You are ready to register the program with *XManager*. Make sure this program is a blocking widget by *not* using the *No_Block* keyword. Making the program a blocking widget program will allow it to be used at the IDL command line without a valid group leader identified for it. Type:

```
XManager, 'openimage', tlb, Event_Handler='OpenImage_Events'
```

Returning from the Block

At this point in program execution, IDL has stopped executing the code in the widget definition module. All the program “action” is taking place in the event handler module. IDL will not come back and finish executing the code in the widget definition module until the blocked widget program is destroyed by the user clicking either the *Cancel* or the *Accept* button. When that happens, there will be dialog information stored in the pointer location.

The idea is to go get this information and process it in any way you like, before you return something to the user as the result of this function. In this case, we are not going to do anything in particular except set the *Cancel* flag. Then we will return the form information to the user. (Another way to have written this program would be to open and read the image data file here, and return the actual image to the user. What you do and what you return is entirely up to you.)

First, go recover the dialog information from the pointer location. You are now done with the pointer, so you can destroy it. Type this:

```
 fileInfo = *ptr  
Ptr_Free, ptr
```

Next, set the *cancel* flag. Type this:

```
cancel = fileInfo.cancel
```

All that is left is to return the file information collected by the dialog form widget to the caller of the program. Type these commands:

```
RETURN, fileInfo  
END
```

You can find the final *OpenImage* program code, including this widget definition module, in the IDL Source Code appendix on page 428.

Writing the Modal Dialog Event Handler Modules

You are ready now to write the event handler modules for this modal dialog form widget program. Let's start with the *OpenImage_BrowseFiles* event handler.

The idea of a *Browse* button is to give the user the ability to select a file name with the mouse instead if having to type the name into a text widget, which is hard to do correctly for almost all of us. *Dialog_Pickfile* is a built-in IDL command to do this for us. All we have to do is test whether the user actually selected a file name. If they did, we simply place that file name in the appropriate text field in our program. The entire code of the *OpenImage_BrowseFiles* event handler is here. Place this code somewhere ahead of the *OpenImage* module in your program file.

```
Pro OpenImage_BrowseFiles, Event
filename = Dialog_Pickfile(Filter='*.dat')
IF filename EQ "" THEN RETURN

Widget_Control, event.top, Get_UValue=info, /No_Copy
Widget_Control, info.fileID, Set_Value=filename
Widget_Control, event.top, Set_UValue=info, /No_Copy
END
```

Notice that we are only going to look for files with a *.dat* file extension, as a result of using the *Filter* keyword with *Dialog_Pickfile*.

The basic idea for the other event handler, *OpenImage_Events* is quite simple. You are only concerned with the *Cancel* and *Accept* button events. Any other events that come into the event handler will simply be ignored. (The *CW_Field* widgets will generate events if the user hits a carriage return in them. Those events should be ignored in this event handler.) If the user clicks the *Cancel* button, you will destroy the widget, thereby releasing the blocking action. You will do the same thing if the user clicks the *Accept* button, but you will also collect the information from the form and store it in the global pointer location before the widget is destroyed.

The first few lines of the event handler module will look like this. (Be sure to add this event handler module to the *openimage.pro* program file in front of the widget definition module.)

```
Pro OpenImage_Events, event
eventName = Tag_Names(event, /Structure_Name)
IF eventName NE 'WIDGET_BUTTON' THEN RETURN
```

Notice the use of the *Tag_Names* command to screen out any event structure that is not a button event structure.

All right, it is a button event you are dealing with if you get to this part of the code. Get the *info* structure and find out which button it was that caused the event. Type this:

```
Widget_Control, event.top, Get_UValue=info, /No_Copy
Widget_Control, event.id, Get_Value=buttonValue
```

If this is the *Cancel* button all you need to do is destroy the widget. This is because the pointer was initialized with information appropriate for a *Cancel* event (i.e., the *Cancel* field of the pointer structure was initialized to 1). Type this:

```
CASE buttonValue OF
'Cancel' : Widget_Control, event.top, /Destroy
```

If this is the *Accept* button, you have to gather information from the widget and store it in the pointer location before you destroy the widget. The code for the *Accept* button and for the rest of the event handler module looks like this:

```
'Accept' : BEGIN
```

```

Widget_Control, info.fileID, Get_Value=filename
filename = filename[0]
Widget_Control, info.xsizeID, Get_Value=xsize
Widget_Control, info.ysizeID, Get_Value=ysize
(*info.ptr).filename = filename
(*info.ptr).xsize = xsize
(*info.ptr).ysize = ysize
(*info.ptr).cancel = 0
Widget_Control, event.top, /Destroy
END
ENDCASE
END

```

Notice that the *filename* variable is subscripted before it is stored in the pointer. This is because what is returned from a text widget is always a string *array*, even if there is only one string in the text widget. We want to make sure we put a scalar value in the pointer location.

Notice, too, how the pointer structure is subscripted. The parentheses around the pointer de-references are essential to access the fields of the structure that are stored in the pointer. This is because a structure de-reference has a higher order of precedence than a pointer de-reference. The parentheses force the pointer de-reference to occur before the structure de-reference, which is exactly what we want in this case.

Finally, notice that the *info* structure does not have to be put back into the user value of the top-level base, since the top-level base is going to be destroyed in any case.

Error Handling

It is a hard and fast rule of widget programming that gathering information of any kind from a user is filled with danger. Users can't spell, they can't (or won't) read directions or program documentation, and they are completely incapable of providing accurate information. In short, it is a miracle that anyone, let alone a prestigious University, gave them an advanced scientific degree. But, uh, they *will* be using your programs. So you better be prepared for them.

Sometimes you can anticipate errors users might make, sometimes (even as cynical as you are) you are surprised by them. In any case, your program should catch and handler all errors with aplomb. We need a *Catch* error handler here. (See “The Catch Control Statement” on page 230 for additional information.)

One error I can anticipate is that the user will delete all the text in one of the text widgets and then hit the *Accept* button. I'm not sure if this will cause an error in the program (I'd have to read the *CW_Field* documentation to find out, and I really don't have time for that!), but I can imagine it might and I'd like to be prepared for it. A little research shows that this particular error is likely be noticed when an undefined variable is used in an expression.

For example, if I get a value out of the X size widget and try to store that value, like this:

```

Widget_Control, info.xsizeID, Get_Value=xsize
(*info.ptr).xsize = xsize

```

I will cause an error if the *xsize* variable is undefined. (This is really a hypothetical situation, since I know perfectly well that *CW_Field* does not return undefined variables in this case. It returns a 0, which is even worse in my humble opinion, because it pushes the error off until it is farther from its source and harder to find. But I am going to give up on *CW_Field* in just a moment anyway because of some of its other limitations, so bear with me just a little longer.)

Using an undefined variable in a program is error number -167. So I could trap for this error in particular, and for all other errors in general, by writing a *Catch* error handler like this. Add the code in bold to the *OpenImage_Events* event handler code as the very first line of code after the procedure definition statement.

```
Pro OpenImage_Events, event
Catch, theError
IF theError NE 0 THEN BEGIN
    Catch, /Cancel
    IF !Error_State.Code EQ -167 THEN BEGIN
        ok = Error_Message('A required value is undefined.')
    ENDIF ELSE BEGIN
        ok= Error_Message()
    ENDELSE
    IF N_Elements(info) NE 0 THEN $
        Widget_Control, event.top, Set_UValue=info, /No_Copy
    RETURN
ENDIF
eventName = Tag_Names(event, /Structure_Name)
```

Notice that I used the *Error_Message* command to report the error, which is a program you downloaded to use with this book. (The *Error_Message* program is described in more detail in “Tracing Errors” on page 234.) I also check the *info* structure back into the user value of the top-level base, but only if it is checked out currently. This is absolutely necessary if I want the program to continue running properly.

As long as we are handling errors, it might be a good idea to think of other kinds of errors we could handle right here in the event handler that the user could fix. For example, if the user spelled the name of the file incorrectly, we wouldn’t be able to find the file when we looked for it. We could catch that error and let the user have another chance at it. (I’m not a big proponent of making users continue to do something they just made a mistake doing. It is bad for their self-esteem. But correcting obvious errors seems more like help than a penalty.)

So before we store the information we collect in the pointer location, let’s do some preliminary checking of the information. Make the changes in bold to the *Accept* button’s code:

```
'Accept' : BEGIN
    Widget_Control, info.fileID, Get_Value=filename
    filename = filename[0]
    Widget_Control, info.xsizeID, Get_Value=xsize
    Widget_Control, info.ysizeID, Get_Value=ysize
    dummy = Findfile(filename, Count=theCount)
    IF theCount EQ 0 THEN $
        Message, 'Requested file cannot be found. ' + $
            'Check spelling.', /NoName
    IF xsize LE 0 OR ysize LE 0 THEN $
        Message, 'File sizes must be positive', /NoName
        (*info.ptr).filename = filename
        (*info.ptr).xsize = xsize
        (*info.ptr).ysize = ysize
        (*info.ptr).cancel = 0
        Widget_Control, event.top, /Destroy
    END
ENDCASE
END
```

Testing the Modal Dialog Form Widget Program

Save your program as *openimage.pro*. Compile and test it from the command line like this:

```
IDL> .Compile openimage
IDL> fileinfo = OpenImage()
```

It should look like the illustration in Figure 117.

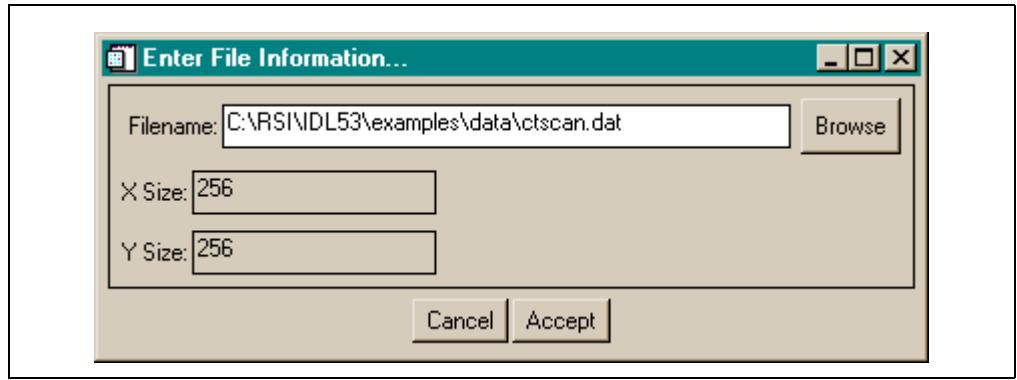


Figure 117: The OpenImage modal dialog form widget program. Notice that the X Size and Y Size text widgets do not look editable in this Windows version of CW_Field. This can be confusing to users.

Notice that the background of the *Filename* text widget appears white and editable. But the fields of the *X Size* and *Y Size* text widgets appear grayed out and do not look editable, even though the user may change these values. This occurs only on Windows platforms, but it can be terribly confusing to the user. Notice too that the *Filename*, *X Size*, and *Y Size* titles do not line up properly. What I would like is for the three editable text widgets to align under one another for aesthetic reasons, with the titles on the left arranged appropriately beside them. There is no way to do this using *CW_Field*.

(The reason the *X Size* and *Y Size* text widgets do not appear editable is that, in fact, they are *not* editable. That is to say, the *Editable* keyword is turned off for these text widgets. The event handler, however, is receiving all the events you type. So that if you type an integer, which is what you said should go into the text field, IDL will place the proper integer character in the field. This gives the appearance of editability, but allows the code to screen character input. A nice idea with an unfortunate consequence.)

Note that when you select either the *Cancel* or *Accept* button, the widget disappears. The form information is returned to you as a result of the function. Type this:

```
IDL> Help, fileinfo, /Structure
```

Using FSC_InputField for Program Input

To get around the appearance problem (many of my users run IDL on Windows platforms) and because I prefer widgets with more aesthetic capabilities, I re-wrote *CW_Field* and named the program *FSC_InputField*. This program is among the programs you downloaded to use with this book. (An added benefit of *FSC_InputField* is that you have a series of fields, as you do here, *FSC_InputField* has the ability to use the *Tab* key to tab from one field to another. This is another feature lacking in *CW_Field*.)

To use *FSC_InputField*, find these four lines in the *OpenImage* widget definition code:

```
fileID = CW_Field(filebase, Title='Filename:', $  
    Value=filename, XSize=filesize)  
browseID = Widget_Button(filebase, Value='Browse', $  
    Event_Proc='OpenImage_BrowseFiles'  
xsizeID = CW_Field(subbase, Title='X Size:', $  
    Value=xsize, /Integer)  
ysizeID = CW_Field(subbase, Title='Y Size:', $  
    Value=ysize, /Integer)
```

Replace the *CW_Field* lines with these:

```
fileID = FSC_InputField(filebase, Title='Filename:', $  
    Value=filename, XSize=filesize, LabelSize=50, $  
    /StringValue)  
browseID = Widget_Button(filebase, Value='Browse', $  
    Event_Proc='OpenImage_BrowseFiles'  
xsizeID = FSC_InputField(subbase, Title='X Size:', $  
    Value=xsize, /IntegerValue, LabelSize=50, Digits=4)  
ysizeID = FSC_InputField(subbase, Title='Y Size:', $  
    Value=ysize, /IntegerValue, LabelSize=50, Digits=4)
```

Notice the *LabelSize* keywords. This sets the label portion of the compound widget to be the same size for all three fields, thus aligning their text widgets. The *Digits* keyword for the integer fields permits the integer to have only four digits. This is another feature missing in *CW_Field* and helps prevent user errors.

The unusual feature about the *FSC_InputField* compound widget, however, is that instead of returning a widget identifier, it returns an object reference. Objects are heap variables, like pointers. The huge advantage of writing compound widgets as objects is that they then become much more versatile in how they can be manipulated after they are created. For example, we can set up a tabbing sequence for these compound widgets by calling the *SetTabNext* method to tell the widget which other widget to go to when a *Tab* character is detected. (The *GetTextID* method returns the widget identifier of the text widget in each compound widget.) For example, type this code immediately after the code above:

```
fileID->SetTabNext, xsizeID->GetTextID()  
xsizeID->SetTabNext, ysizeID->GetTextID()  
ysizeID->SetTabNext, fileID->GetTextID()
```

The only other changes you need to make in the *OpenImage* program are in the two event handler modules. First of all, you need to set the value of the file name text widget differently. Find this code in the *OpenImage_BrowseFiles* event handler:

```
Widget_Control, event.top, Get_UValue=info, /No_Copy  
Widget_Control, info.fileID, Set_Value=filename  
Widget_Control, event.top, Set_UValue=info, /No_Copy
```

Change the code like this:

```
Widget_Control, event.top, Get_UValue=info, /No_Copy  
info.fileID->Set_Value, filename  
Widget_Control, event.top, Set_UValue=info, /No_Copy
```

And in the *OpenImage_Events* event handler, where you are responding to the *Accept* button, find these lines of code:

```
'Accept' : BEGIN  
    Widget_Control, info.fileID, Get_Value=filename
```

```

filename = filename[0]
Widget_Control, info.xsizeID, Get_Value=xsize
Widget_Control, info.ysizeID, Get_Value=ysize

```

Modify the code to obtain the text widget values from the objects like this. (Make the changes in bold.)

```

'Accept' : BEGIN
    filename = info.fileID->Get_Value()
    filename = filename[0]
    xsize = info.xsizeID->Get_Value()
    ysize = info.ysizeID->Get_Value()

```

Save and re-compile the program. It should now look similar to the illustration in Figure 118.

```

IDL> .Compile openimage
IDL> fileinfo = OpenImage()

```

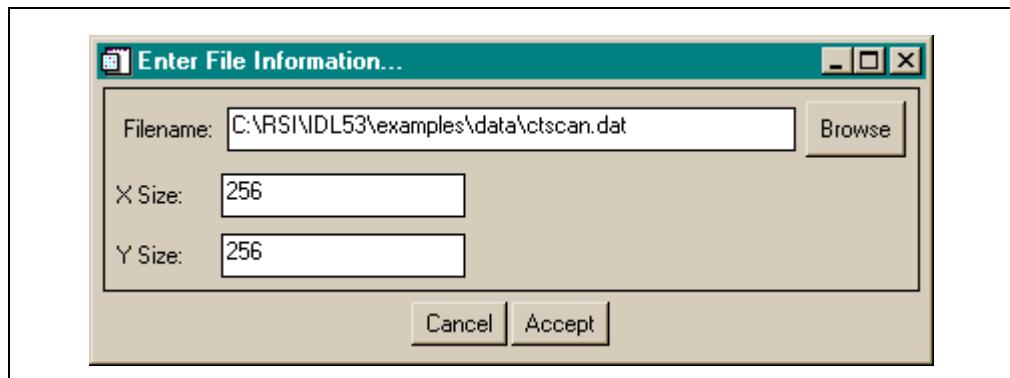


Figure 118: The *OpenImage* program with FSC_Inputfield compound widgets in place of CW_Field compound widgets. The text widgets now look editable and the three fields align under one another. Users can also tab from one field to another, a capability not present in CW_Field.

Try the program and see how different it is from the one you had working previously. Try tabbing from one field to another. What happens if you eliminate the value in, say, the XSize field and then select the Accept button? Is this what you expect?

You can find a copy of the *OpenImage* program among the programs you downloaded to use with this book. It is named *openimage.pro*.

Adding an Open Image Capability to the Histo_GUI Program

Let's test the *OpenImage* program by adding an *Open Image* functionality to the *Histo_GUI* program we were working on in the last chapter. If you didn't create the *Histo_GUI* program, you can use the file *histo_gui.6.pro* that you downloaded to use with this book as the starting point for the programming exercises in this chapter. Rename the program file *histo_gui.pro*.

Adding an Open Button

The first step is to add an *Open* button under the *File* menu and before the *Print* button. I like to add an ellipsis to the *Open* button as a visual clue to the user that the button will cause another widget program (in this case, the modal dialog widget *OpenImage*) to appear. Add the line (in bold) to the *Histo_GUI* widget definition module just after the line that creates the *File* button:

```
fileID = Widget_Button(menuBarID, Value='File')
openID = Widget_Button(fileID, Value='Open...', $
    Event_Pro='Histo_GUI_Open_Image')
```

The event handler for this *Open* button will be named *Histo_GUI_Open_Image*.

Writing the Open Image Event Handler

As always, we can write the skeleton of the *Histo_GUI_Open_Image* event handler, like this. Add this procedure somewhere in front of the widget definition module in the file.

```
PRO Histo_GUI_Open_Image, event
Widget_Control, event.top, Get_UValue=info, /No_Copy
Widget_Control, event.top, Set_UValue=info, /No_Copy
END
```

The purpose of this event handler is to ask the user to supply some information about an image file. Then, using that information, you are going to open the file for reading and read the image data from the file. If that goes successfully, you will then modify the *info* structure to be pointing to the correct image and re-display it.

There are many, many things in this scenario that can go badly wrong. You could collect lousy information from the user, either because the user can't type, or because the user just doesn't know the answers to the questions you are asking. If you have bad information, it will be impossible to read the data correctly. Sometimes you will be able to read the data from the file, but it will be the wrong data. We can recover from the first kind of error, but we cannot recover (or even *discover* in a program) the second kind of error.

In any case, we have to provide an event handler that doesn't break just because something goes wrong. The only way to do this is to catch and handle both the expected and the unexpected errors in a graceful way. We will do this with a *Catch* error handler.

The following code will go into the event handler just after the procedure definition line and before we get the *info* structure from the user value of the top-level base:

```
PRO Histo_GUI_Open_Image, event
Catch, theError
IF theError NE 0 THEN BEGIN
    Catch, /Cancel
    ok = Error_Message(Traceback=1)
    IF N_Elements(info) NE 0 THEN $
        Widget_Control, event.top, Set_UValue=info, /No_Copy
    RETURN
ENDIF
```

We can actually do quite a lot of work before we need the *info* structure. For example, we can collect the information from the user about what image file to read and the size of the image data, and we can even read the image data, all before we need the *info* structure.

The next piece of code will collect the file name and image sizes from the user. Remember we want the *OpenImage* program to be a modal widget program, so we must use the *Group_Leader* keyword when we call it. The *Cancel* keyword will be checked to see if the user cancelled from the program. Remember that the cancel flag will be set if the *OpenImage* program encounters an error, also. The code will look like this:

```

 fileInfo = OpenImage(Cancel=cancelled, $
 Group_Leader=event.top)
 IF cancelled THEN RETURN

```

When the *OpenImage* program is destroyed, the keyboard focus will return to the *Histo_GUI* program. This will cause a re-draw of the program in the window. Since this is really not necessary in this case (we haven't changed any colors, for example), we may wish to turn keyboard focus events off while we have the *OpenImage* program on the display. If that is the case, you can add this code (in bold) to the two lines you just typed:

```

Widget_Control, event.top, KBRD_Focus_Events=0
 fileInfo = OpenImage(Cancel=cancelled, $
 Group_Leader=event.top)
Widget_Control, event.top, KBRD_Focus_Events=1
 IF cancelled THEN RETURN

```

Now you have (presumably) the information you need to read the image file. Let's do so, the code (placed immediately below the lines above) looks like this:

```

newimage = BytArr(fileInfo.xsize, fileInfo.ysize)
OpenR, lun, fileInfo.filename, /Get_Lun
ReadU, lun, newimage
Free_Lun, lun

```

Remember, we are assuming the file contains a 2D byte array.

Now that we have the new image, we need to store it in the *info* structure. So here is where we obtain the *info* structure. (This line of code is already in the file.)

Storing the new image is as simple as pointing the image pointers to the new image. You do not have to worry at all about de-allocating or freeing the previous pointer. IDL takes care of all the memory management for you. The code looks like this:

```

Widget_Control, event.top, Get_UValue=info, /No_Copy
*info.image = newimage
*info.process = newimage
*info.undo = newimage

```

Since we can't undo this operation, we should make the *Undo* button insensitive. Add this line of code.

```
Widget_Control, info.undoID, Sensitive=0
```

All that remains is to re-display the graphics. The final code looks like this. (Add the lines in bold type.)

```

WSet, info.pixID
HistoImage, *info.process, $
 AxisColorName=info.axisColorName, $
 BackColorName=info.backcolorName, $
 Binsize=info.binsize, $
 DataColorName=info.datacolorName, $
 _Extra=*info.extra, $
 Max_Value=info.max_value, $
 NoLoadCT=1, $
 XScale=info.xscale, $
 YScale=info.yscale
WSet, info.wid
Device, Copy=[0, 0, !D.X_Size, !D.Y_Size, 0, 0, info.pixID]
Widget_Control, event.top, Set_UValue=info, /No_Copy

```

END

Save the file, re-compile it, and test it. (If you did not enter the program above into a file named *histo_gui.pro*, you can use the file *histo_gui.7.pro*, which you downloaded to use with this book. This file must be explicitly compiled to be used. See “The Advantage of Mistakes” on page 261 for more information. The program as written is also listed in *Appendix C: IDL Program Code* as the *Histo_GUI* program. See page 428.)

Does the program work as you expect? Try to generate some errors. This shouldn’t be hard, since you probably know nothing about the files in the IDL distribution. If you want to know more about files that you might possibly try to open, you can find file descriptions in “Appendix B: Data File Descriptions” on page 397. What happens if you open a file that is not on that list? What happens if you mis-spell a filename or type an incorrect file size?

Creating a Non-Modal Widget Dialog

Sometimes you don’t want everything to stop while you collect information from the user. You would prefer to have the dialog on the display for the convenience of the user, but you don’t want it to block the use of other programs that may also be on the display. For example, the *XColors* program that you are using to change image colors in the *Histo_GUI* program works in this way. This is a much more problematic situation because you don’t know *when* the user will be ready for you to process the information. The dialog could be on the display for hours before the user chooses to use it.

In this situation you have to devise some way for the dialog form widget to notify another program that the user is ready for something to happen. I usually implement this by means of an *Apply* button, which is the equivalent to the *Accept* button from the previous programming exercise. Similarly, I have a *Dismiss* button which is the equivalent to the *Cancel* button from the previous exercise. Its purpose is to destroy the widget without doing anything else. In all other respects, the non-modal dialog form widget will look identical to the modal version. For example, compare the non-blocking dialog widget in the illustration in Figure 119, which is the program *ReadImage* that you are about to write, with the *OpenImage* program in Figure 118.

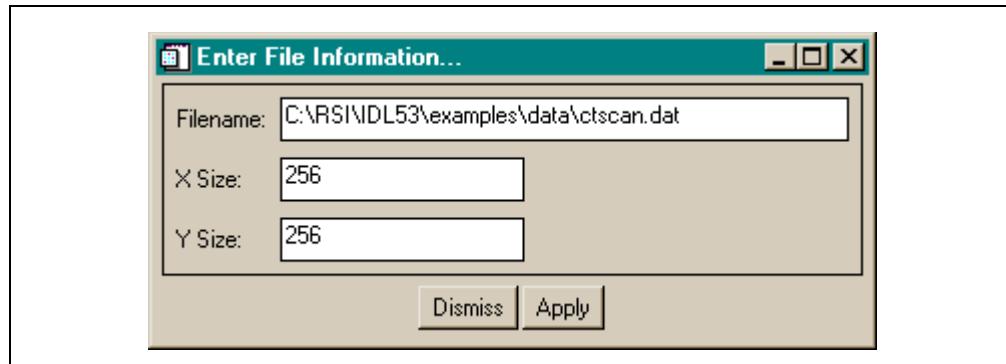


Figure 119: The non-modal dialog form widget program *Readimage*. Notice that it looks almost identical to the modal dialog form widget program *OpenImage* in Figure 117. Only the button labels are a clue to its non-blocking status.

Writing a Non-Modal Dialog Widget Definition Module

The widget definition module for the *ReadImage* program is almost identical to the widget definition module for the *OpenImage* program you wrote earlier. However, there are a few important exceptions. First, the program is written as a procedure rather than as a function. Since there is nothing really to return (the *ReadImage* program will remain on the display until the user selects the *Dismiss* button), there is no need to make it a function. Second, no pointer is used. We will not be storing any data anywhere. We will collect and send the form data to the user of the program when the user selects the *Apply* button. The “sending” of the data will be via a widget event structure. Third, we do not want to make the widget program a modal or blocking widget program. On the contrary, we specifically want it to be a non-blocking widget program. We will make it so by using the *No_Block* keyword on the *XManager* command.

Here is the entire code of the *ReadImage* widget definition module. The changes from the previous *OpenImage* program are indicated by bold characters.

```

PRO ReadImage, $
    notifyIDs, $
    filename=filename, $
    Group_Leader=group_leader, $
    XSize=xsize, $
    YSize=ysize

    On_Error, 2 ; Return to caller.

    IF N_Elements(notifyIDs) EQ 0 THEN $
        Message, 'Notification IDs are a required parameter.'
    IF N_Elements(filename) EQ 0 THEN $
        filename=Filepath('ctscan.dat', $
            SubDirectory=['examples','data'])
    IF N_Elements(xsize) EQ 0 THEN xsize = 256
    IF N_Elements(ysize) EQ 0 THEN ysize = 256

    tlb = Widget_Base(Column=1, /Base_Align_Center, $
        Title='Enter File Information...')

    subbase = Widget_Base(tlb, Column=1, Frame=1)
    filebase = Widget_Base(subbase, Row=1)

    fileID = CW_Field(filebase, Title='Filename:', $
        Value=filename, XSize=filesize)
    browseID = Widget_Button(filebase, Value='Browse', $
        Event_Pro='ReadImage_BrowseFiles')
    xsizeID = FSC_InputField(subbase, Title='X Size:', $
        Value=xsize, /IntegerValue, LabelSize=50, Digits=4)
    ysizeID = FSC_InputField(subbase, Title='Y Size:', $
        Value=ysize, /IntegerValue, LabelSize=50, Digits=4)

    fileID->SetTabNext, xsizeID->GetTextID()
    xsizeID->SetTabNext, ysizeID->GetTextID()
    ysizeID->SetTabNext, fileID->GetTextID()

    butbase = Widget_Base(tlb, Row=1)
    cancel = Widget_Button(butbase, Value='Dismiss')
    accept = Widget_Button(butbase, Value='Apply')

    screenSize = Get_Screen_Size()
    geom = Widget_Info(tlb, /Geometry)
    Widget_Control, tlb, $

```

```

XOffset = (screenSize[0] / 2) - (geom.scr_xsize / 2), $
YOffset = (screenSize[1] / 2) - (geom.scr_ysize / 2)

Widget_Control, tlb, /Realize
info = { notifyIDs:notifyIDs, $
         fileID:fileID, $
         xsizeID:xsizeID, $
         ysizeID:ysizeID }

Widget_Control, tlb, Set_UValue=info, /No_Copy
XManager, 'readimage', tlb,
Event_Handler='ReadImage_Events', $
/No_Block, Group_Leader=group_leader
END

```

Notifying Widgets of Program Events

The most important change to this code occurs on the first line of the *ReadImage* widget definition module. The code has changed from a function to a procedure:

```

PRO ReadImage, $
  notifyIDs, $
  Filename=filename, $
  Group_Leader=group_leader, $
  XSize=xsize, $
  YSize=ysize

```

The reason for this change is that you will not be able to return the information as the return value of the function, since the program will not be pausing or blocking while the user fills out the dialog form. You must find another technique to notify the calling program (or any other program) that the information is available and ready to be recovered.

The technique used here is to accept, via the *notifyIDs* argument, a vector of widget identifiers. When the user clicks the *Apply* button, the information on the form will be collected and the widgets identified by the *notifyIDs* argument will be notified by sending each widget an event structure containing the form data.

The *notifyIDs* argument will be a 2-by-*n* array of widget identifiers. The first column will identify the widgets to be notified when the *Apply* button is selected. The second column will identify the top-level base widgets associated with the widgets in the first column. In practice, the *ReadImage* program will usually be called from within an event handler module like this:

```
ReadImage, [event.id, event.top], Group_Leader=event.top
```

The next change in the widget definition module is to make the *notifyIDs* parameter a required argument:

```

IF N_Elements(notifyIDs) EQ 0 THEN $
  Message, 'Notification IDs are a required parameter.'

```

The way the top-level base widget is defined is also changed:

```

tlb = Widget_Base(Column=1, /Base_Align_Center, $
                  Title='Enter File Information...')

```

Notice that this is not a modal widget, so there is no use of *Modal* or *Floating* keywords. The *Group_Leader* keyword could be used here, but it is not. I prefer to assign the group leader with the *Group_Leader* keyword on the *XManager* command, but this is strictly a personal preference.

Although the names of the widget buttons have changed to *Dismiss* and *Apply*, their functions are similar to the functions of the *Cancel* and *Accept* buttons in the *GetData* program:

```
dismissID = Widget_Button(butbase, Value='Dismiss')
applyID = Widget_Button(butbase, Value='Apply')
```

Changing the button names is one more visual cue to the user about the function of this widget. In other words, users who see *Cancel* and *Accept* buttons should expect modal functionality. While users who see *Dismiss* and *Apply* buttons should expect non-modal functionality. Users will be confused less often if you keep your interface as consistent as possible.

There is no pointer required in this program because there is no need for a global memory location. But the vector of widget identifiers (identifying those widgets that are to be notified of an event) must be stored in the *info* structure, since these will be needed in the event handler module:

```
info = { notifyIDs:notifyIDs, $
         fileID:fileID, $
         xsizeID:xsizeID, $
         ysizeID:ysizeID }
```

The final change in this widget definition module is to make the widget a non-blocking widget by using the *No_Block* keyword on the *XManager* command:

```
XManager, 'readimage', tlb,
    Event_Handler='ReadImage_Events', $
    /No_Block, Group_Leader=group_leader
```

Notice that this is where I assign the group leader for the top-level base.

Writing the Non-Modal Dialog Event Handler Modules

There will be no difference in how the file browsing event handlers in the *OpenImage* and the *ReadImage* programs work. You can simply copy the event handler from the *OpenImage* program you created before and rename it in this program.

```
Pro ReadImage_BrowseFiles, Event
filename = Dialog_Pickfile(Filter='*.dat')
IF filename EQ "" THEN RETURN
Widget_Control, event.top, Get_UValue=info, /No_Copy
Widget_Control, info.fileID, Set_Value=filename
Widget_Control, event.top, Set_UValue=info, /No_Copy
END
```

The biggest difference between the *OpenImage* and *ReadImage* programs is in how the *ReadImage_Events* event handler module works when the user hits the *Accept* or *Apply* button. The rest of the event handler is much the same. Here is the first half of the event handler code, with the minor differences shown in bold type:

```
Pro ReadImage_Events, event
eventName = Tag_Names(event, /Structure_Name)
IF eventName NE 'WIDGET_BUTTON' THEN RETURN
Widget_Control, event.top, Get_UValue=info, /No_Copy
Widget_Control, event.id, Get_Value=buttonValue
CASE buttonValue OF
    'Dismiss' : Widget_Control, event.top, /Destroy
```

Sending Events to Other Widgets

When the user clicks the *Apply* button, the event handler will collect the information from the dialog form and notify the widgets specified by the *notifyIDs* parameter. This is done not by putting the file information into a pointer storage location as before, but by creating a *READIMAGE_EVENT* event structure and sending it to the widget with the *Send_Event* keyword to the *Widget_Control* command. The second half of the event handler code (with differences in bold type) looks like this:

```
'Apply' : BEGIN
    filename = info.fileID->Get_Value()
    filename = filename[0]
    xsize = info.xsizeID->Get_Value()
    ysize = info.ysizeID->Get_Value()

    dummy = Findfile(filename, Count=theCount)
    IF theCount EQ 0 THEN $
        Message, 'Requested file cannot be found. ' + $
            'Check spelling.', /NoName
    IF xsize LE 0 OR ysize LE 0 THEN $
        Message, 'File sizes must be positive', /NoName

    s = Size(info.notifyIDs)
    IF s[0] EQ 1 THEN count = 0 ELSE count = s[2] - 1
    FOR j=0,count DO BEGIN
        fileInfo = { READIMAGE_EVENT, $
                    ID:info.notifyIDs[0,j], $
                    Top:info.notifyIDs[1,j], $
                    Handler:0L, $
                    Filename:filename, $
                    XSize:xsize, $
                    YSize:ysize }
        IF Widget_Info(info.notifyIDs[0,j], /Valid_ID) THEN $
            Widget_Control, info.notifyIDs[0,j], $
            Send_Event=fileInfo
    ENDFOR
    Widget_Control, event.top, Set_UValue=info, /No_Copy
END
ENDCASE
END
```

The *fileInfo* event structure that gets created is a named event structure with the name *READIMAGE_EVENT*. The *ID* field contains the identifier of the widget that will receive the event. The *Top* field identifies the corresponding widget at the top of the widget hierarchy that contains the *ID* widget. The *Handler* field is empty, but this will be filled out correctly by IDL itself before the event structure gets to the *ID* widget. All you need to do here is create the field as a long integer. The *Filename*, *XSize*, and *YSize* fields contain information collected from the dialog form widget. This is the information that previously was stored in the pointer.

If the widget that should receive this event structure is still a valid widget (i.e., it still exists), then the *fileInfo* event structure is sent to it with the *Widget_Control* command and the *Send_Event* keyword. This is a widget event just like any other widget event. In other words, it goes into the event queue and is dispatched to the proper widget in exactly the same way as if we had selected a button widget from the graphical user interface.

Notice that the widget is *not* destroyed by selecting the *Apply* button. This widget will remain on the display until the *Dismiss* button is selected. Since the widget program is

not destroyed, it is essential to put the *info* structure back in its storage location in the user value of the top-level base before exiting this event handler.

The *ReadImage* program is among the programs you downloaded to use with this book. Its name is *readimage.pro*. The complete program listing for the *ReadImage* program is also available in “Appendix C: IDL Program Code” on page 431.

Testing the ReadImage Program

To test the *ReadImage* program, we have to modify the *Histo_GUI* program slightly. Since you might prefer—as I do—that the *Open* button call a modal dialog form widget (e.g., *OpenImage*), let’s make the changes to a copy of the *Histo_GUI* program as it was written before the *Open* button was added to it.

Open the file *histo_gui.6.pro* in your text editor. This file is among the program files you downloaded to use with this book. The *Histo_GUI* program in this file is as written at the end of the previous chapter. Re-name the file *histo_gui_nonmodal.pro*. We will make the following program changes to this file. (Alternatively, you can look at the file *histo_gui.8.pro*, which has the changes I am about to describe already added to it. This file is among the files you downloaded to use with this book.)

The *histo_gui_nonmodal.pro* program file will require several modifications. Namely, we will have to define the *Open* button, and we will have to write the event handler for the button.

The *Open* button is defined in the *Histo_GUI* widget definition module, just after the *File* button is defined. Add the code in bold, like this:

```
fileID = Widget_Button(menuBarID, Value='File')
openID = Widget_Button(fileID, Value='Open...', $
Event_Pro='Histo_GUI_Read_Image')
```

The event handler for this *Open* button will be named *Histo_GUI_Read_Image*.

Writing the Read Image Event Handler

As always, we can write the skeleton of the *Histo_GUI_Read_Image* event handler, like this. Add this procedure somewhere in front of the widget definition module in the file.

```
PRO Histo_GUI_Read_Image, event
Widget_Control, event.top, Get_UValue=info, /No_Copy
Widget_Control, event.top, Set_UValue=info, /No_Copy
END
```

The purpose of this event handler is to ask the user to supply some information about an image file. Then, using that information (whenever it arrives), you are going to open the file for reading and read the image data from the file. If that goes successfully, you will then modify the *info* structure to be pointing to the correct image and re-display it.

As before, there are many things that can go wrong. We will trap all possible errors with a *Catch* error handler. The following code will go into the event handler just after the procedure definition line and before we get the *info* structure from the user value of the top-level base:

```
PRO Histo_GUI_Read_Image, event
Catch, theError
IF theError NE 0 THEN BEGIN
    Catch, /Cancel
    IF !Error_State.Code EQ -167 THEN BEGIN
```

```

        ok = Error_Message('A required value is undefined.')
ENDIF ELSE BEGIN
    ok= Error_Message()
ENDELSE
IF N_Elements(info) NE 0 THEN $
    Widget_Control, event.top, Set_UValue=info, /No_Copy
RETURN
ENDIF
Widget_Control, event.top, Get_UValue=info, /No_Copy

```

At this point the code will begin to resemble the *Histo_GUI_Image_Colors* event handler we wrote previously. (See “Writing the Image Colors Event Handler” on page 305.) That is to say, we expect two different kinds of events in this event handler: a *WIDGET_BUTTON* event if the *Open* button was selected, and an event sent from the *ReadImage* program, named *READIMAGE_EVENT*, alerting us to the fact that the user selected the *Apply* button on that program. We wish to branch on the structure name. The *CASE* statement code will look like this and will immediately follow the line above in which the *info* structure is retrieved from the user value of the top-level base.

```

thisEvent = Tag_Names(event, /Structure_Name)
CASE thisEvent OF
    'WIDGET_BUTTON': BEGIN
    END

    'READIMAGE_EVENT': BEGIN
    END

ENDCASE
Widget_Control, event.top, Set_UValue=info, /No_Copy
END

```

Of course, if this is a button event, we want to call the *ReadImage* program. That code, placed in the proper place in the *CASE* statement, looks like this:

```
'WIDGET_BUTTON': BEGIN
    ReadImage, [event.id, event.top], Group_Leader=event.top
END
```

Notice that the widget we want notified is the *Open* button itself. That is how we will get back into this event handler when the user selects the *Apply* button on the *ReadImage* program. We use the *Group_Leader* keyword to assign the *Histo_GUI* top-level base as the group leader for the *ReadImage* program. We do this so that the *ReadImage* program will be destroyed when the *Histo_GUI* program is destroyed.

If this is an event from the *ReadImage* program, then the code will closely resemble the code you wrote for the *OpenImage* event handler. The only difference is that you will be receiving the file information from an event structure, rather than from a structure that was returned from the *OpenImage* function. This portion of the *CASE* statement will look like this:

```
'READIMAGE_EVENT': BEGIN
    newimage = BytArr(event.xsize, event.ysize)
    OpenR, lun, event.filename, /Get_Lun
    ReadU, lun, newimage
    Free_Lun, lun
    *info.image = newimage
    *info.process = newimage

```

```

*info.undo = newimage
Widget_Control, info.undoID, Sensitive=0
WSet, info.pixID
HistoImage, *info.process, $
    AxisColorName=info.axisColorName, $
    BackColorName=info.backcolorName, $
    Binsize=info.binsize, $
    DataColorName=info.datacolorName, $
    _Extra=*info.extra, $
    Max_Value=info.max_value, $
    NoLoadCT=1, $
    XScale=info.xscale, $
    YScale=info.yscale
WSet, info.wid
Device, Copy=[0, 0, !D.X_Size, !D.Y_Size, 0, 0, info.pixID]
END

```

Save and compile this program. Run it like this:

```

IDL> .Compile histo_gui_nonmodal.pro
IDL> Histo_GUI

```

Does the program work like you expect it to? If you need information about data files found in the IDL distribution, you can find the information in “Appendix B: Data File Descriptions” on page 397.

What happens if you select the *Open* button in the *Histo_Gui* program several times? Whoops! This is probably not what you had in mind. Every time you select the button a new *ReadImage* program appears.

We can limit the user to only one *ReadImage* program at a time by checking to see if a program with the name *readimage* is already registered with *XManager*. If it is, we will just return out of the *ReadImage* widget definition module. Make this change (in bold) to the *ReadImage* program module, just after the procedure definition statement:

```

PRO ReadImage, $
    notifyIDs, $
    Filename=filename, $
    Group_Leader=group_leader, $
    XSize=xsize, $
    YSize=ysize

IF XRegistered('readimage') NE 0 THEN RETURN

```

Re-compile and run your program again. (If you did not enter the program above into a file named *histo_gui_nonmodal.pro*, you can use the file *histo_gui.8.pro*, which you downloaded to use with this book. This file must be explicitly compiled to be used. See “The Advantage of Mistakes” on page 261 for more information.) This is better, I think.

Chapter 13

F Discovering the Possibilities FFF



Creating Graphics Display Objects

Chapter Overview

The purpose of this chapter is to demonstrate how to build graphics display objects. These are objects that use direct graphics commands for graphics display. We often think of these objects as “smart” objects or “sticky” objects since their properties are persistent. This is unlike most IDL graphics display commands, whose properties are ephemeral. Among the techniques you will learn are these:

- How object oriented design principles are implemented in IDL
- How objects are created in IDL
- How to write initialization and clean-up lifecycle methods
- How to write object methods
- The meaning of such words as encapsulation, inheritance, and polymorphism

A Quick Object Overview

There has been a great deal of talk and excitement in the past several years over “object-oriented” programming. To hear some people tell it, object-oriented programs are going to rule the world, making our lives easier, longer, and more fulfilling. Uh, huh. And some day pigs will fly.

But, still.... So much buzz can’t just be for naught. What is it about object-oriented programming that has people so excited? Does it make programs easier to write? Do they work better? Faster? Are object programs easier to maintain than they used to be?

The answer, frankly, is yes and no. I’m not sure object programs are easier to write, although they are often much easier to change and modify than other programs. I’m not sure they are much easier to maintain, although they do tend to keep functionality separated into small chunks of code. In truth, objects are just another programming tool that is available to you. What you make of objects probably has more to do with you and your skills as a programmer, than it does with objects or object-oriented programming per se.

When objects were first introduced in IDL 5, I was surprised to learn that I had been writing and teaching object-oriented programming practices for a number of years. For example, the way we have written the *Histo_GUI* program in the past several chapters is an object-oriented approach to programming. Each small event handler can

be thought of as an object *method*, a way of interacting with the data in the program. The *info* structure can be thought of as the very heart of the object itself, being the place where the data and the information required to work with the data resides. In some sense, objects just confirm and extend our notions about what a properly written IDL program looks like.

The Idea of Data Encapsulation

Be that as it may, the basic idea of objects is that data and the procedures and functions (called *methods* when applied to objects) that can be used to work with or manipulate that data can be encapsulated into a black-box-like shell, called the *object*. The abstraction of the black-box—its definition, if you like—is called the object’s *class*. In other words, when we speak of an object of a particular *class*, we have in mind an object with particular properties: *this* kind of data and *these* kinds of methods that can be used to manipulate that data. When we actually load the data into the object, and configure the object with certain parameters (much as you would set up fields in an *info* structure), then we are dealing with a specific *instance* of the object. In other words, the *instance object* is a particular object, with specific properties, of a general class of objects, with the class’s generalized properties.

For example, we might imagine an object class named *Family_Dog*. As a class, it is composed of a living animal that requires food and water, moderate exercise, someone to scratch its ears, build it a dog house, etc. A particular instance of that object class is *Sage*, a female Golden retriever, two years old, light brown in color, who jumps all over you in greeting, with an 11 year old boy to scratch her ears, and who still can’t—after all this time—resist chewing the odd shoe left unprotected in the family room. (As you gaze at her lying under your feet, it suddenly occurs to you that she has inherited the object class *Beloved_Family_Member*, but that’s another story, told later in this chapter.)

Once data has been encapsulated inside an object, the only way to interact with that data is through the object’s methods. In other words, the data is shielded from view to anyone not using an object method to access the data. Methods are just IDL procedures and functions of the normal type. The only difference is that they only have effect or scope, if you like, inside the object. In other words, they can only be used to work with, or manipulate, the object’s data.



Note that IDL’s implementation of data encapsulation in objects is not perfect. It won’t take most programmers ten minutes to figure out how it can be defeated. But don’t do it. You will completely destroy both the purpose and integrity of object programming by allowing object data to be accessible outside of the object’s methods. To make full use of objects, they must be used appropriately.

IDL objects are *heap variables*, just as pointers are heap variables. That is, they exist somewhere in IDL’s global memory space. This makes objects persistent in the same way pointers are persistent. An object reference is a light-weight (four-byte) token, pointing to something much more extensive, that exists until the object is explicitly destroyed.

Creating Objects

Objects are created by using the *Obj_New* command to create an object. For example, the programs you downloaded to use with this book contain a PostScript configuration object with a class name *FSC_PSCConfig*. (The file is named *fsc_psconfig_define.pro*. Notice the two underscore characters before the word *define* in the file name. This is important, as you will see in just a moment.) A specific instance of this object class can be created by typing this command:

```
IDL> thisObject = Obj_New('FSC_PSCConfig')
```

Notice that the first (and only, in this case) parameter to the *Obj_New* command is the class name of the object. If you look, you will see that the variable *thisObject* is an object reference. Type this command:

```
IDL> Help, thisObject
```

You will see something similar to this:

```
THISOBJECT      OBJREF      = <ObjHeapVar3 (FSC_PSCONFIG) >
```

This tells you the object reference is a heap variable. Indeed, you can check the heap and see what is there. Your output should look similar to this.

```
IDL> Help, /Heap
```

```
Heap Variables:
# Pointer: 1
# Object : 1
```

```
<ObjHeapVar3>   STRUCT     = -> FSC_PSCONFIG Array [1]
<PtrHeapVar4>   STRING     = Array [10]
```

There is one object reference, which appears to be a structure, and one pointer reference on the heap. Where did the pointer come from? Although you don't know it yet, the pointer undoubtedly came from the object, since pointers are convenient ways to save data inside an object. In fact, it would be impossible to have objects without having pointers. You will learn more about this in just a moment.

Invoking Object Methods

This particular object is used to configure a PostScript device. It has a number of methods that allow you to do so. Remember that methods are just normal IDL procedures and functions, but they can only be used to manipulate data that is encapsulated inside an object. The author of the object class will publish or document the methods that go with the object. Normally, we would look in the object's class definition file (*fsc_psconfig_define.pro*, in this case) for the documentation for the object's methods. However, since there are so many methods associated with this object class, I've made a special web page to describe the methods:

```
http://www.dfanng.com/programs/docs/fsc\_psconfig.html
```

One of the methods you see described there is a *GUI* method. This method puts up a graphical user interface that allows the user to configure the PostScript device in any way that suits him or her. An object method is invoked with an "arrow" operator, which is just a hyphen followed by the greater-than sign, like this:

```
IDL> thisObject->GUI
```

Like any other procedure or function in IDL, object methods can also be passed positional and keyword arguments. For example, hit the *Cancel* button on the graphical user interface on your display, and re-invoke the *GUI* method like this:

```
IDL> thisObject->GUI, Cancel=cancelled, /NoBlock
```

Note that calling the *GUI* method is no different from calling a *GUI* procedure, except that the object reference and the arrow operator are in front of the name of the method or procedure.

Play with some of the settings for a moment. When you have the configuration the way you like it, hit the *Accept* button. The current configuration is now saved inside the object. If you hit the *Dismiss* button to destroy the graphical user interface and

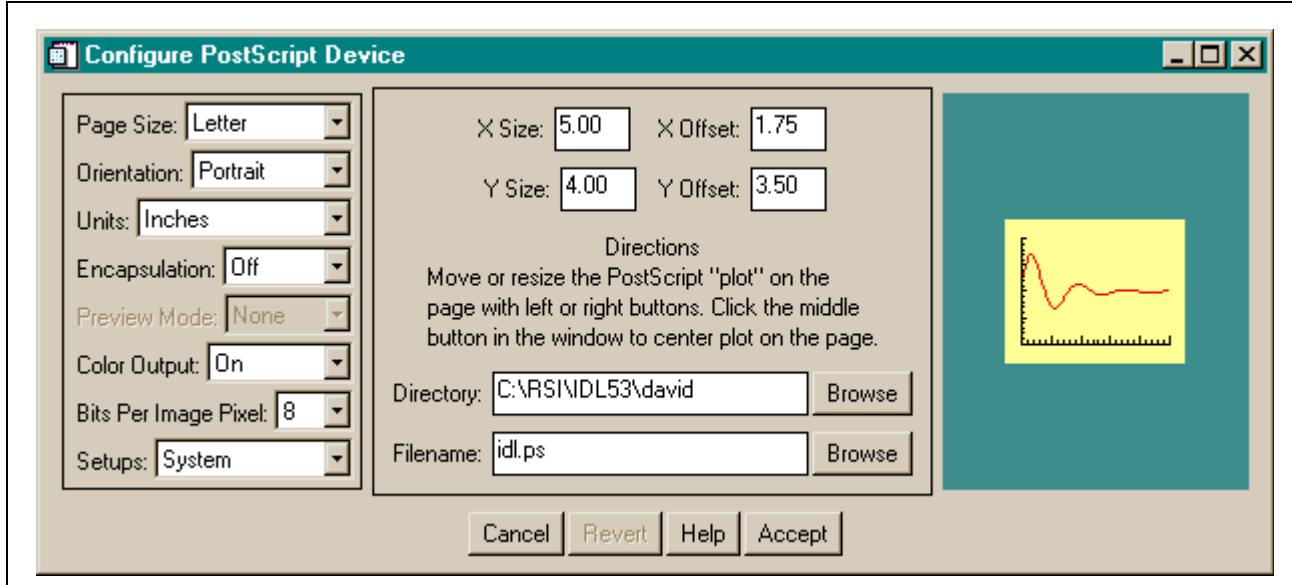


Figure 120: The result of applying the *GUI* method to the *FSC_PSCConfig* object.

then invoke the *GUI* method again, the graphical user interface comes up with the last settings you had accepted before you dismissed it. This type of persistent or remembered behavior is one of the great benefits of objects.

In order to configure the PostScript device with the object, we need to be able to get the configuration values out of the object. There are several ways to do this, but the easiest is to just get a structure of the appropriate “device” keywords that we can pass to the PostScript device using the keyword inheritance mechanism. We can do this by invoking a function method on the object named *GetKeywords*, like this:

```
IDL> psKeywords = thisObject->GetKeywords()
```

Notice that in this case the *GetKeywords* method contains no arguments, but the parentheses indicate it is a function. (It is similar to the function *N_Params* in this sense.) You can see what the return value of the function method is by typing this:

```
IDL> Help, psKeywords, /Structure
```

You see something similar to this, where each field in the structure is the name of a keyword that can be used to control the PostScript device.

```
** Structure <1569bd0>, 24 tags, length=68, refs=1:
BITS_PER_PIXEL    INT          8
COLOR             INT          1
ENCAPSULATED     INT          0
FILENAME          STRING       'C:\RSI\IDL53\coyote\idl.ps'
FONT_SIZE         INT          12
INCHES            INT          1
ISOLATIN1         INT          0
PREVIEW           INT          0
TT_FONT           INT          0
XOFFSET           FLOAT        1.50000
XSIZE             FLOAT        8.00000
YOFFSET           FLOAT        9.50000
YSIZE             FLOAT        5.50000
LANDSCAPE         INT          1
PORTRAIT          INT          0
```

HELVETICA	INT	1
BOLD	INT	0
BOOK	INT	0
DEMI	INT	0
ITALIC	INT	0
LIGHT	INT	0
MEDIUM	INT	0
NARROW	INT	0
OBLIQUE	INT	0

In practice, the PostScript device is configured with the *FSC_PSCConfig* object like this:

```
Set_Plot, 'PS'
Device, _Extra=thisObject->GetKeywords()
```

Destroying Objects

As heap variables, objects persist in the IDL environment until they are explicitly destroyed. Objects are destroyed (and removed from the heap) with the *Obj_Destroy* command, like this:

```
IDL> Obj_Destroy, thisObject
```

Creating a New Object Class

I think it will be instructive to construct a display object that is similar to the programs *HistoImage* and *Histo_GUI* you wrote in the last several chapters. That way you can see both the similarities and the differences in the two programming styles. I want to do this in several steps, however, because one of the advantages of object programming is that objects can be re-used and combined to produce other programs. In other words, most people construct object libraries that can be used over and over again in their programming projects. This approach will allow us to see how this concept can be put into practice.

I'd like to start by creating a *BoxImage* object class. This object class will be able to display an image with axes around the image. We will have the ability to assign scales (values) to the axes and to label them appropriately. A color bar can be displayed either horizontally or vertically with the image.

Defining the Object Class

The first step in creating a new object is to define the object class. Objects are implemented in IDL as named structures. This has both advantages and disadvantages to the IDL programmer. One major advantage is that most IDL programmers are already familiar with structures. For example, most IDL widgets generate named event structures, and you used an anonymous *info* structure to pass information around in the widget programs you wrote in the past several chapters.

But for those of you who are not quite sure about them, here is a short review.

Structure Review

Structures are variables that hold heterogeneous data. Data in structures can be distinguished by its logical relationships, rather than because its form or type is the same. A named structure is a structure that has a name. Here is a named structure, that has the name *Employee*:

```
IDL> struct = {Employee, BadgeID:0L, Age:0, Salary:0.0}
```

There are three fields in this structure: *BadgeID*, a long integer; *Age*, an integer; and *Salary*, a floating value. It groups information about an “employee” together in a single variable.

The statement above is what I like to call a *formal structure definition statement*. That is to say, the *Employee* structure is defined in a formal sense by that statement, since the data type and size of each field is filled out explicitly.

More often, especially in your colleague’s code, you will see what I call an *informal structure definition statement*. It might look like this:

```
IDL> struct = {Employee, BadgeID:58637L, Age:29, $  
    Salary:47598.0}
```

In this statement, we have not only defined the structure *Employee*, we have filled it with specific data, which is stored in the variable *struct*.

It doesn’t matter whether you have defined the structure formally or informally, IDL saves the *definition* of the named structure internally. It does not save the specific data of the structure in the structure definition. Saving the definition of the structure makes it easy to create another of those *Employee* structures. You do it like this:

```
IDL> joe = {Employee}  
IDL> joe.badgeID = 498325L  
IDL> joe.age = 34  
IDL> joe.salary = 34,568.0
```

In this case, the variable *joe* is one of those *Employee* structures defined above. Notice that I can immediately start to fill the fields of the structures, because IDL already knows those fields are defined and what they are defined as. In fact, I can even create a specific instance of the *Employee* structure in a variable *mary* like this:

```
IDL> mary = {Employee, 594832L, 26, 45346.0}
```

As long as the values and order of the input matches the definition and order of the defined fields in the *Employee* structure, IDL is content to let me define a particular instance of a structure this way.

What if sometime later I want to add a picture of the employee to the structure? I might try to re-define the structure like this:

```
IDL> struct = {Employee, BadgeID:0L, Age:0, Salary:0.0, $  
    Picture:BytArr(300,300)}
```

IDL doesn’t let me do this and an error to the effect that you have defined the “wrong number of tags” results. This may or may not alert you to what you are doing wrong. But, take my word for it, you cannot extend a named structure this way. The only way to do this is to exit IDL (for all versions of IDL prior to 5.3) or to use the *.Reset_Session* executive command to reset the entire IDL session. Doing this will completely remove all main-level program variables, all compiled procedures and sessions, and all named common blocks, as well as removing all current named structure definitions. Be sure you know the consequences of your action.

```
IDL> .Reset_Session  
IDL> struct = {Employee, BadgeID:0L, Age:0, Salary:0.0, $  
    Picture:BytArr(300,300)}
```

Since objects are implemented in IDL as named structures, you find the *.Reset_Session* command immensely helpful during object program development!

Automatic Structure Definition

Many programmers are aware of automatic command definition in IDL. For example, if I type the command *Junk* at the IDL command line:

```
IDL> Junk
```

IDL doesn't have any idea what I mean by that and begins to search in its directory path for a file named *junk.pro*. If it finds such a file, it compiles the program modules in the file until it either comes to a program module of the right name (*junk*, in this case) or encounters a main-level program. In either case, IDL immediately stops compiling and runs the module.

What many programmers are not aware of is automatic structure definition in IDL.

For example, if I had not typed the structure definition above after resetting the IDL session and I had typed this line instead:

```
IDL> mary = {Employee, 594832L, 26, 45346.0, maryPicture}
```

IDL could conceivably have been confused about what to do. In fact, what it would do is try to find a file containing the definition of the structure *Employee*, so it could use it in this expression. It would look for such a definition in a file *employee_define.pro*, where there are two underscore characters in a row in front of the word *define* in the file name.

Had we written such a file, like this:

```
PRO Employee_Define
struct = {Employee, BadgeID:0L, Age:0, Salary:0.0, $
          Picture:BytArr(300,300)}
END
```

Then IDL would have had no trouble creating a specific instance of this structure named *mary*. This is called automatic structure definition and is analogous to automatic command definition in IDL.



Note that if I had defined the structure in the *Employee_Define* module above with actual data, that the data would not be a part of the structure definition. It is similar to what happens in an informal structure definition statement. When IDL calls this program module all it is looking for is the *definition* of the structure. It could care less what the actual values of the fields are.

The BoxImage Class Definition

Such a structure definition module is at the heart of every object you create in IDL. In fact, it is this structure definition module that defines the object class. The name of the object class is identical to the name of the named structure.

The object class definition structure will hold the data that is to be encapsulated in the object. (In this sense, it is like the *info* structure in the *Histo_GUI* program you wrote in the last several chapters.) In this case, the data will include image data, positions in the window, scales for the axes, colors for annotation and background display, image colors, etc.

Open a new file from your editor and name the file *boximage_define.pro*. Type the following structure definition, which in this case is the *BoxImage* object class definition:

```
PRO BoxImage_Define
struct = { BOXIMAGE, $ ; The object class name.
          image: Ptr_New(), $ ; The image data.
          process: Ptr_New(), $ ; The processed image.
```

```

        undo: Ptr_New(), $      ; The undo image.
        position: FltArr(4), $ ; Position in window.
        r: Ptr_New(), $       ; Vector of red indices.
        g: Ptr_New(), $       ; Vector of green indices.
        b: Ptr_New(), $       ; Vector of blue indices.
        ncolors: OL, $        ; Number of image colors.
        annotatecolor: "", $ ; Annotation color name.
        backcolor:"", $       ; Background color name.
        xscale: FltArr(2), $ ; Scale for X axis.
        yscale: FltArr(2), $ ; Scale for Y axis.
        xtitle: "", $         ; Title for X axis.
        ytitle: "", $         ; Title for Y axis.
        vertical: OL, $       ; Vertical colorbar flag.
        extra: Ptr_New() }    ; Extra keywords.

END

```

Notice this is a structure definition, *not* the actual data. IDL calls this routine *only* to get the object class definition. You will populate this object with actual data in the next step. Since this is a named structure, and since we have no way of knowing ahead of time about how some of the fields will actually be defined (Is the *image* field going to be a 300 by 300 byte array, or a 512 by 440 float array?), we define all fields that will have changing or variable size or type data to a null pointer type. In other words, later this field will be a real pointer to some real data. At the moment we are just saving enough space in our structure (object) definition for a pointer data type.

This is the program module that will get called when you create an object of class *BoxImage*. (But please don't type this command yet. We want to initialize the object properly, and that will be difficult if we create the object now. The reasons for this will be explained in the next section.)

```
myObject = Obj_New('BoxImage')
```

In other words, by typing the command above, you ask IDL to create an object reference to a class of objects named *BoxImage*. IDL knows that objects are implemented as named structures, so it goes looking for a file with the name *boximage_define.pro*. If and when it finds it (we just wrote it) IDL compiles the file until it finds a module with the same name, in which case it stops compiling and runs that module, thereby providing a definition for the *BoxImage* object class.

You can think of this as a formal definition of the object class, if you like, in a way that is analogous to the formal definition of a structure.

But, a specific object does get created. The object named *myObject* in the command above is a specific instance of an object of class *BoxImage*, with specific (formalized) object data. Most of the time we would prefer the object to have other, particular, data. The question is, how does an object get assigned that particular data? Most of the time, it does so by automatically calling an *Init* method for the *BoxImage* object class.

Creating Object Lifecycle Methods

There are two lifecycle methods: *Init* and *Cleanup*. Lifecycle methods are special because they cannot be called in the way normal object methods are called. Rather, they are called automatically when an object is created or destroyed, respectively.

For example, when IDL executes this command, not only is the *BoxImage_Define* module compiled and run to define the object class, but the *Init* method of the *BoxImage* object class is called to initialize the object with specific data.

```
myObject = Obj_New('BoxImage')
```

- + Objects do not have to be written with *Init* and *Cleanup* methods, but in practice almost all objects are. If the methods are available, they are called when the object is created or destroyed.

Creating the Init Method

The *Init* method is a *function*. If you have a red pen, get it out and underline that last sentence. You will inevitably make your *Init* method a procedure the first three or four times you create an object. If you don't get totally discouraged when your objects never work and give up object programming entirely, you might remember I told you that the *Init* method must be a function. It is critically important.

Let's see. Did I mention that the *Init* method must be a function? If so, it bears repeating. Your objects won't work if you write the *Init* method as a procedure.

Here is why. The purpose of the *Init* method is to return a 1 if the *Init* method works properly. It is suppose to return a 0 if anything at all goes wrong. Procedures always return an implicit 0 to the caller. This means that if you make the *Init* method a procedure, IDL will think your *Init* method failed every time and you will never get a valid object reference. Please, please, please make it a function. (Alright, enough pleading. It's just that every time I go look over the shoulder of a new, struggling object programmer they have made the *Init* function a proc... Oh, never mind. I'm sure you have the idea by now.)

The *Init* method function can be written just exactly like any other IDL function. You can have any number of positional and keyword arguments. Those arguments can be input or output parameters, etc. The only thing different about this function is the way you specify its name. It *must* be named (no exceptions) with the object class name, followed by two colons, and then the word *Init*. It cannot be named *Initialize*, or anything else. It must be called *Init*.

Open up the *boximage_define.pro* file and add the *Init* method code *in front of* the object definition module (*BoxImage_Define*) in the file. You do this for the very same reason that you add event handler code in front of the widget definition module code in a widget program file: because you want all the helper modules to be compiled properly before they have to be used.

- + Note that methods can be put into their own individual files, although I don't know anyone who does this (except inexperienced programmers). If you choose to do this, the file should be named with two underscore characters instead of two colons. For example, you could put the *Init* method in a separate file name *boximage_init.pro*.

The function definition statement of the *Init* method looks like this:

```
Function BoxImage::Init, $ ; The name of the method.
    image, $ ; The image data.
    AnnotateColor=annotatecolor, $ ; The annotation color.
    BackColor=backcolor, $ ; The background color.
    ColorTable=colortable, $ ; The colortable index.
    NColors=ncolors, $ ; Number of image colors.
    Position=position, $ ; Position in window.
    Vertical=vertical, $ ; Vertical colorbar flag.
    XScale=xscale, $ ; The scale on X axis.
    XTitle=xtitle, $ ; The title on X axis.
    YScale=yscale, $ ; The scale on Y axis.
    YTitle=ytitle, $ ; The title on Y axis.
    _Extra=extra ; Holds extra keywords
```

Notice the way the *Init* method name is composed. You must use the object class name, two colons, and the name of the method. Here we define a single positional

parameter: the image to be displayed, and a number of keyword parameters that will set properties for this object.

Recall that the *Init* method is to return a 0 if anything goes wrong. In practice, you usually *Catch* all errors in the *Init* method and return a 0 if an error occurs. The error handling code might look like this:

```

Catch, theError
IF theError NE 0 THEN BEGIN
    Catch, /Cancel
    ok = Error_Message(!Error_State.Msg + ' Returning...', $
        Traceback=1, /Error)
    RETURN, 0
ENDIF

```

As in all procedures and functions, the next step is check to be sure all the positional and keyword parameters have values. The code will look like this:

```

IF N_Elements(image) EQ 0 THEN image = LoadData(7)
ndims = Size(image, /N_Dimensions)
IF ndims NE 2 THEN $
    Message, 'Image must be 2D array.', /NoName
    ; Check for keyword parameters.

IF N_Elements(annotationcolor) EQ 0 THEN $
    annotationcolor = "NAVY"
IF N_Elements(backcolor) EQ 0 THEN backcolor = "WHITE"
IF N_Elements(ncolors) EQ 0 THEN ncolors = !D.Table_Size - 3
IF N_Elements(position) EQ 0 THEN $
    position = [0.15, 0.15, 0.9, 0.9]
vertical = Keyword_Set(vertical)
IF N_Elements(xtitle) EQ 0 THEN xtitle = ""
IF N_Elements(ytitle) EQ 0 THEN ytitle = ""

s = Size(image, /Dimensions)
IF N_Elements(xscale) EQ 0 THEN xscale = [0,s[0]]
IF N_Elements(yscale) EQ 0 THEN yscale = [0,s[1]]

```

There is nothing unusual here. We will provide an image if one isn't passed into the program. We will have a white background color and a navy annotation color. The axes scales will be taken from the size of the image if nothing is provided.

The *Colortable* keyword gives us a bit more challenge. In the *HistoImage* program we wrote previously (see the “The HistoImage Program” on page 240 for additional information) we chose to actually load a color table. This gave us problems when we wanted to control colors from outside the program as in the *Histo_GUI* program and required us to define a *NoLoadCT* keyword to prevent color table loading. The persistence of objects gives us another way to handle this problem.

In this program, what we want to do is store the actual RGB vectors that contain the color indices of the 2D image. But to do that properly, we need to resample the actual color table vectors, which are always 256 elements in length, into the number of colors we are going to use to display the image. Getting those color vectors without loading the color table is the trick we need here.

Fortunately, IDL has provided an answer in the form of the IDL graphics object library, and the *IDLgrPalette* object in particular. We can create the palette object, load the color table indicated by the user into the palette with its *LoadCT* method, then get the RGB vectors from the palette object to use in this object. The *GetProperty* method of the *IDLgrPalette* object will allow us to retrieve the currently loaded color

vectors via output keywords. In this way, no color table vectors ever have to be loaded into the physical color table of the device. The code to do so will look like this:

```

IF N_Elements(colortable) EQ 0 THEN BEGIN
    colors = Obj_New("IDLgrPalette")
    colors->LoadCT, 0
    colors->GetProperty, Red=r, Green=g, Blue=b
    Obj_Destroy, colors
ENDIF ELSE BEGIN
    colors = Obj_New("IDLgrPalette")
    colors->LoadCT, 0 > colortable < 40
    colors->GetProperty, Red=r, Green=g, Blue=b
    Obj_Destroy, colors
ENDELSE

r = Congrid(r, ncolors)
g = Congrid(g, ncolors)
b = Congrid(b, ncolors)

```

Notice that if the user does not specify a color table, we will display the image in a gray-scale color table. Notice, too, that the *IDLgrPalette* object is destroyed when we are finished with it. This is so we don't have any leaking heap memory in the program.

Now, recall that I said the purpose of the *Init* method was to initialize or populate with specific data a particular instance of the object. That particular instance of the object, inside each of the object's methods, is a variable named *self*. You don't have to declare it. It is just there, automatically, in each object method. It is that particular instance of that particular object class. And inside the methods you can refer to that *self* object as if it were a structure, defined as in the object class definition. This is the only place you can do so. (You can also refer to the *self* object as an object reference, which it really is, and call methods on it, and so forth.)

So, the next step is to populate the *self* object. We refer to the structure definition statement, and populate the *self* object like this:

```

self.image = Ptr_New(image)
self.process = Ptr_New(image)
self.undo = Ptr_New(image)
self.position = position
self.ncolors = ncolors
self.annotatecolor = annotatecolor
self.backcolor = backcolor
self.r = Ptr_New(r)
self.g = Ptr_New(g)
self.b = Ptr_New(b)
self.vertical = vertical
self.xscale = xscale
self.yscale = yscale
self.xtitle = xtitle
self.ytitle = ytitle
self.extra = Ptr_New(extra)

```

Note that the *extra* variable will be created by IDL if "extra" keywords are passed into the program via the *_Extra* keyword inheritance mechanism. Extra keywords are those keywords that are not specifically defined for the *Init* function. I am thinking, in particular, that users might want to set keywords for the *TVImage*, and *Colorbar* commands that I will eventually use in a display method I still have to write. I want to have a place to store those keywords. If the *extra* variable is undefined at this point, then no extra keywords were used, and *self.extra* is a pointer to an undefined variable.

Recall that this is a valid type of pointer (unlike the null pointer, for example, that is an invalid pointer), and that it can be dereferenced in the normal way.

If we get to this place in the *Init* function code, we have successfully populated the object with data. The only thing to do now is return a 1 to indicate success. The code looks like this:

```
RETURN, 1  
END
```

Creating the Cleanup Method

The second lifecycle method is the *Cleanup* method. Like the *Init* method, which is called automatically when the object is created, and cannot be called directly from an object, the *Cleanup* method is called automatically when the object is destroyed and cannot be called directly from an object.

The purpose of an object *Cleanup* method, like the purpose of a widget program's *Cleanup* routine, is to clean up or destroy those items such as pointers, objects, pixmap windows, etc. that must be handled properly to prevent memory leakage. In this case, the *Cleanup* method must be concerned with freeing the seven pointer variables used in the object. You want to add the *Cleanup* module, which will be written as a procedure with no parameters, to your code somewhere before the *BoxImage_Define* module. (The very start of the *boximage_define.pro* file is a good place.) The complete code of the *Cleanup* method looks like this.

```
PRO BoxImage:::Cleanup  
  
Ptr_Free, self.image  
Ptr_Free, self.process  
Ptr_Free, self.undo  
Ptr_Free, self.r  
Ptr_Free, self.g  
Ptr_Free, self.b  
Ptr_Free, self.extra  
  
END
```

Creating the Specific Instance of the Object

Once the object class definition (*BoxImage_Define*), *Init*, and *Cleanup* modules are defined, it is possible to create an object. Compile your program and make sure you have no program errors. Fix any that occur.

```
IDL> .Compile boximage_define
```

Next, try creating an object:

```
IDL> theObject = Obj_New('BoxImage')
```

Did you get any errors from the *Init* method? If so, fix them and try again. If something goes wrong, then your object will be a null reference. For example, if you have an error in your *Init* method code and the *Init* method returned a 0 instead of a 1, then when you perform a *Help* on the object reference you will see something like this:

```
IDL> Help, theObject  
theObject          OBJREF      = <NullObject>
```

If you created the object successfully, you should see something similar to this:

```
IDL> Help, theObject  
theObject          OBJREF      = <ObjHeapVar4(BOXIMAGE)>
```

Lifecycle Methods Must Be Defined When the Object is Created



Pay particular attention to the next sentence; the information it contains is not in the IDL documentation and you can only discover it for yourself by frustrating hours of work. The lifecycle methods *must be* defined at the time when the first object of this class is created in the IDL session if you want them to be “attached to” or “associated with” this object. If you create the object without either the *Init* method or *Cleanup* method being defined for the object, then all subsequent objects you create in that IDL session (regardless of how many times you write the lifecycle methods or compile the program code) will be bereft of those lifecycle methods.

If for some reason you create the initial object without having written the *Init* or *Cleanup* methods, then you will have to either re-start IDL (if you have versions of IDL prior to IDL 5.3) or use the *.Reset_Session* executive command to reset your entire IDL session to get those methods associated with the objects you create.

Common Problems When Creating Objects

If you have any problems at all creating your object, try these suggestions:

- 1 Check to be sure your *Init* method is written as a function and not as a procedure.
- 2 Check to be sure you are returning a 1 from your *Init* function.
- 3 Exit IDL and re-start or reset the IDL session with *.Reset_Session*. Try your object creation routine now. Early versions of IDL objects required not only that the *Init* method be written prior to object creation, but that it function properly (without errors) too. Fix your code, reset the session, and try it again.
- 4 The three steps above will fix 99 percent of all the errors in creating an object successfully, but if you still are having problems examine your code carefully to be sure it is doing what you think it is doing. And, in particular, make sure every module is compiling when you compile the code.

If you got your object to initialize properly, then you will want to destroy it before proceeding. Recall that you use the *Obj_Destroy* command to destroy an object.

```
IDL> Obj_Destroy, theObject
```

This is when you will discover any problems with your *Cleanup* method. If you have problems, fix your code, recompile and try to destroy the object again.

Make sure your heap is free of any object or pointer references before proceeding. Type this:

```
IDL> Help, /Heap
```

If you see any object or pointer references, destroy or free any you have that are current. If none are current, then these references have leaked memory. You can clean the heap memory up with the *Heap_GC* command.

```
IDL> Heap_GC
```

Remember that *Heap_GC* is a last ditch solution and you should not use it in your code, but only from the IDL command line. Well written programs shouldn’t leave anything on the heap to clean up.

Initializing the Object Using Parameters

I mentioned earlier that the *Init* method is called when the object is created, so you might imagine that the time to use the positional and keyword parameters defined for the *Init* method is at the time the object is created. You would be right. For example, if you want to load the *BoxImage* object with the earth elevation data set, and display the image with color table 5, with axes scales ranging from -1 to 1, then you can create the object like this:

```
IDL> earth = LoadData(7)
IDL> thisObject = Obj_New('BoxImage', earth, Colortable=5, $
                           XScale=[-1,1], YScale=[-1,1])
```

When this command is executed, the positional and keyword parameters are passed directly to the *Init* method as soon as the object class definition module has been executed. Only if the *Init* method returns a 1 is the object officially “created”.

Unfortunately, we have no way of seeing if our intended modifications of the *BoxImage* object have had any effect. We need to build some way to display the data in a window.

Creating the Display Method

The traditional name for a display method is *Draw*, but the method doesn’t have to have this name. You can give it any name you like. With the exception of the two life-cycle methods, *Init* and *Cleanup*, which must have those names, you are free to name the methods you create for objects anything you like.

The purpose of the *Draw* method, of course, is to display the graphical output. Like the *HistoImage* program you wrote earlier (see “The *HistoImage* Program” on page 240 for more information), you want to write the display portion of the code in a device independent and color decomposition independent way.

The procedure definition statement, and the *Catch* error handler that should go into most of the object methods you write, will look like this. Add the code to the program file somewhere before the last *BoxImage__Define* program module.

```
PRO BoxImage::Draw, Font=font
      ; Error handling.

      Catch, theError
      IF theError NE 0 THEN BEGIN
          Catch, /Cancel
          ok = Error_Message(!Error_State.Msg + ' Returning...', $
                             Traceback=1, /Error)
          RETURN
      ENDIF
```

The *Font* keyword is used primarily to print the annotation in different kinds of fonts, depending upon which device we are using. It should be set to *!P.Font* by default. Type this:

```
IF N_Elements(font) EQ 0 THEN font=!P.Font
```

Next, we want to load the background, annotation, and image colors. As before, you will use the *GetColor* program you downloaded to use with this book to load the drawing colors in a device and color decomposition state independent way. Type this:

```
annotateColor = GetColor(self.annotatecolor, $
                         !D.Table_Size-2)
backColor = GetColor(self.backColor, !D.Table_Size-3)
TVLCT, *self.r, *self.g, *self.b
```

Next, we want to calculate positions in the window for the image and the color bar. Of course, these positions will depend on whether we are going to have a horizontal color bar (the default) or a vertical color bar. The code will look like this:

```
IF self.vertical THEN BEGIN
    p = self.position
    length = p[2] - p[0]
    imgpos = [p[0], p[1], (p[2]-(0.20*length)), p[3]]
```

```

cbpos = [(p[0]+(0.93*length)), p[1], p[2], p[3]]
ENDIF ELSE BEGIN
  p = self.position
  length = p[3] - p[1]
  imgpos = [p[0], p[1], p[2], (p[3]-(0.20*length))]
  cbpos = [p[0], (p[1]+(0.93*length)), p[2], p[3]]
ENDELSE

```

In the *HistoImage* program we wrote earlier, we were able to set the background color by using the *Background* keyword on the initial *Plot* command. We don't have that luxury here, because we are not going to be issuing commands that typically erase the display window. (We will use *TVImage* and *Colorbar*.) So we are going to have to use the *Erase* command to produce the background color.

But this will cause a problem because *Erase* is not a device-independent command. On the display device *Erase* causes the background to be painted in a specific color, in the PostScript device or the *Printer* device, it produces another page of output. If we always issue an *Erase* command before we issue the graphics commands, we will always get two pages of output (the first one blank) every time we try to create a PostScript file or send graphics output to the printer.

We can solve this problem by only issuing the *Erase* command on those devices that support windows. We can use the *Flags* field of the *!D* system variable to determine if the device supports windows. If the bit representing the number 256 is set, the device supports windows. The code will look like this:

```
IF (!D.Flags AND 256) NE 0 THEN Erase, Color=backColor
```

Next, we want to calculate an appropriate character size for the annotation. As we did in the *HistoImage* program, we will use the *Str_Size* program you downloaded to use with this book to calculate an appropriate character size. Type:

```
thisCharSize = Str_Size('A Sample String', 0.20)
```

Finally, we are ready to draw the image, axes, and color bar. We will use the *Plot* command to draw the axes around the image. The *TVImage* and *Colorbar* programs are among those you downloaded to use with this book. The final code in the draw method will look like this:

```

TVImage, BytScl(*self.process, Top=self.ncolors-1), $
  Position=imgpos, _Extra=*self.extra
Plot, self.xscale, self.yscale, XStyle=1, YStyle=1, $
  XTitle=self.xtext, YTitle=self.ytitle, $
  Color=annotateColor, Position=imgpos, /NoErase, $
  /NoData, Ticklen=-0.025, _Extra=*self.extra, $
  CharSize=thisCharSize, Font=font
Colorbar, Range=[Min(*self.process), Max(*self.process)], $
  Divisions=8, _Extra=*self.extra, Color=annotateColor, $
  Position=cbpos, Ticklen=-0.2, Vertical=self.vertical, $
  NColors=self.ncolors, CharSize=thisCharSize, Font=font
END

```

Save your program file and compile the program. If you have errors, correct them and re-compile.

```
IDL> .Compile boximage_define.pro
```

There is no need to recreate your object, simply use the last one you created with the Earth data set and call the *Draw* method on the object, like this:

```
IDL> thisObject->Draw
```

Your output should look similar to the illustration in Figure 121.

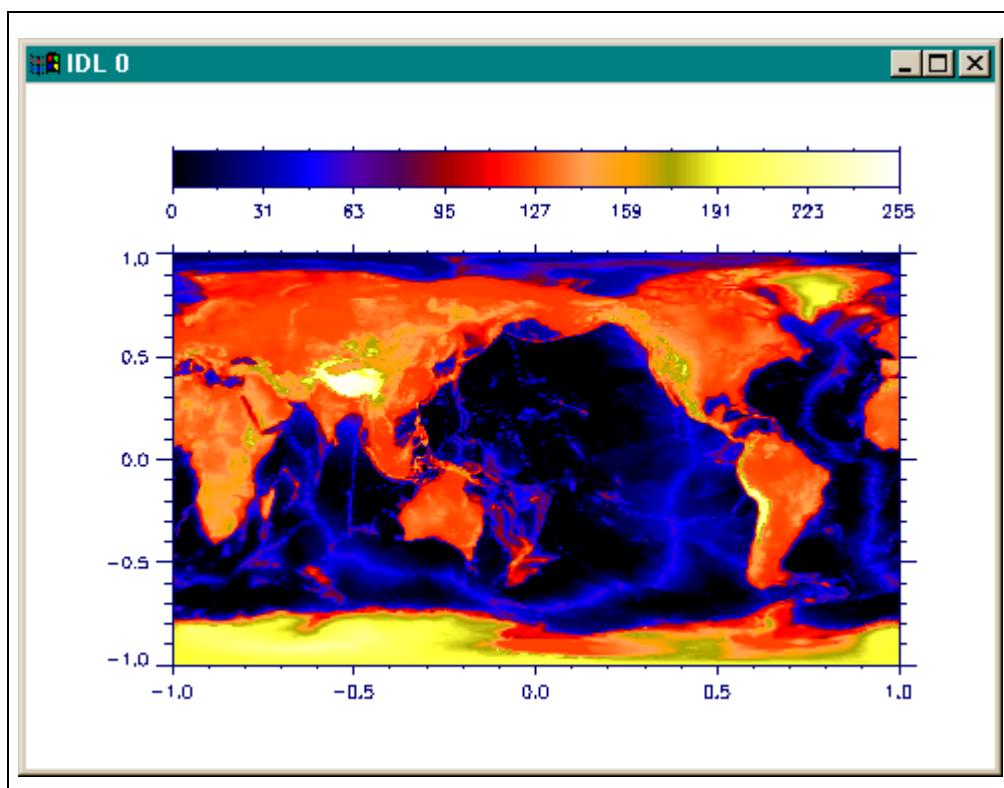


Figure 121: The graphics display after invoking the *BoxImage* Draw method.

Prove to yourself that your program is device independent by, for example, sending the output directly to your default printer. Type these commands:

```
IDL> thisDevice = !D.Name  
IDL> Set_Plot, 'PRINTER', /Copy  
IDL> thisObject->Draw, Font=0  
IDL> Device, /Close_Document  
IDL> Set_Plot, thisDevice
```

You can find a copy of the *BoxImage* object program as it is written so far among the programs you downloaded to use with this book. You must explicitly compile this file in order to use it. The file is named *boximage_define.1.pro*.

```
IDL> .Compile boximage_define.1.pro  
IDL> scan = LoadData(5)  
IDL> thisObject = Obj_New('BoxImage', scan, Colortable=5, $  
XScale=[-1,1], YScale=[-1,1])  
IDL> thisObject->Draw
```

Creating Methods to Set and Get Object Properties

One of the enormous advantages of display objects is that their properties are persistent. That is to say, we don't have to specify a long list of keywords to make the graphics look the way we want them to look. We simply call the *Draw* method on the object and the graphics are displayed appropriately. But what about changing the properties after the object has been created? For example, what if we decide we don't like the look of the graphics output and want the image to be displayed with a different color table?

Set Property Methods

It is easy enough to write a method to do the job for us. In fact, it is a *requirement* that we write a method to do the job. For example, to change to a different color table, we can write a *LoadCT* method. Add this method to your program file somewhere in front of the *BoxImage_Define* module. The code will take advantage of the *IDLgrPalette* object, like we did in the *Init* method. In fact, the code will be almost identical. Type this:

```
PRO BoxImage::LoadCT, colortable
IF N_Elements(colortable) EQ 0 THEN BEGIN
    colors = Obj_New("IDLgrPalette")
    colors->LoadCT, 0
    colors->GetProperty, Red=r, Green=g, Blue=b
    Obj_Destroy, colors
ENDIF ELSE BEGIN
    colors = Obj_New("IDLgrPalette")
    colors->LoadCT, 0 > colortable < 40
    colors->GetProperty, Red=r, Green=g, Blue=b
    Obj_Destroy, colors
ENDELSE
*self.r = Congrid(r, self.ncolors)
*self.g = Congrid(g, self.ncolors)
*self.b = Congrid(b, self.ncolors)
END
```

In this method, we load color table 0 if a color table index number is not provided. Otherwise we load the specified color table. Notice how we scale the color vectors into the number of colors we are using to display the image.

Re-compile your program file, and then try the *LoadCT* method. Type this:

```
IDL> .Compile boximage_define.pro
IDL> thisObject->LoadCT, 3
IDL> thisObject->Draw
```

Notice how you had to call the *Draw* method after you changed the color table to see the new colors in the display window. The *LoadCT* method simply loads new color vectors in the object, it doesn't load them in the device color table, nor does it re-display the graphics.

But since I often want to change a property in my object and see the change go immediately into effect in the graphics window, I like to have the ability to call the *Draw* method from the method that is changing the property. I usually do this by defining a *Draw* keyword for the method that is setting the property. Make the changes in bold to the code you just typed above:

```
PRO BoxImage::LoadCT, colortable, Draw=draw
IF N_Elements(colortable) EQ 0 THEN BEGIN
    colors = Obj_New("IDLgrPalette")
    colors->LoadCT, 0
    colors->GetProperty, Red=r, Green=g, Blue=b
    Obj_Destroy, colors
ENDIF ELSE BEGIN
    colors = Obj_New("IDLgrPalette")
    colors->LoadCT, 0 > colortable < 40
    colors->GetProperty, Red=r, Green=g, Blue=b
    Obj_Destroy, colors
ENDELSE
```

Creating Graphics Display Objects

```
*self.r = Congrid(r, self.ncolors)
*self.g = Congrid(g, self.ncolors)
*self.b = Congrid(b, self.ncolors)

IF Keyword_Set(draw) THEN self->Draw
END
```

With this code in place (re-compile your program file), you can make a change to the color table and see the effect immediately in the graphics window:

```
IDL> thisObject->LoadCT, 33, /Draw
```

While writing a method to set every possible property in the object is possible, it is not usually done this way. Usually, a generic *SetProperty* method is written that can change many properties at the same time via keywords to the method. For example, the procedure definition statement for such a *SetProperty* method could be written like this. Place this code somewhere in front of the *BoxImage_Define* module in the program file.

```
PRO BoxImage:: SetProperty, $
    Image = image, $
    AnnotateColor=annotatecolor, $
    BackColor=backcolor, $
    ColorTable=colortable, $
    Draw=draw, $
    NColors=ncolors, $
    Position=position, $
    Vertical=vertical, $
    XScale=xscale, $
    XTitle=xtitle, $
    YScale=yscale, $
    YTitle=ytitle, $
    _Extra=extra
```

In this method we can change the image itself, all the drawing and image colors, the position of the graphic in the display window, the scales and titles on the axes, etc. We have also defined a *Draw* keyword, which will act like the *Draw* keyword on the *LoadCT* method you just created, allowing us to see the effects of these changes to the object immediately in the display window.

The next step is to write an error handler for this method. Add this code to the file.

```
Catch, theError
IF theError NE 0 THEN BEGIN
    Catch, /Cancel
    ok = Error_Message(!Error_State.Msg + ' Returning...', $
        Traceback=1, /Error)
    RETURN
ENDIF
```

Now, we simply set the object properties if the value of the keyword is defined in the program. The code looks like this:

```
IF N_Elements(backcolor) NE 0 THEN $
    self.backcolor = backcolor
IF N_Elements(annotatecolor) NE 0 THEN $
    self.annotatecolor = annotatecolor
IF N_Elements(ncolors) NE 0 THEN self.ncolors = ncolors
IF N_Elements(position) NE 0 THEN self.position = position
IF N_Elements(vertical) NE 0 THEN self.vertical = vertical
```

```

IF N_Elements(xtitle) NE 0 THEN self.xtitle = xtitle
IF N_Elements(ytitle) NE 0 THEN self.ytitle = ytitle
IF N_Elements(xscale) NE 0 THEN self.xscale = xscale
IF N_Elements(yscale) NE 0 THEN self.yscale = yscale
IF N_Elements(extra) NE 0 THEN *self.extra = extra
IF N_Elements(image) NE 0 THEN BEGIN
    *self.image = image
    *self.process = image
    *self.undo = image
ENDIF
IF N_Elements(colortable) NE 0 THEN BEGIN
    colors = Obj_New("IDLgrPalette")
    colors->LoadCT, 0 > colortable < 40
    colors->GetProperty, Red=r, Green=g, Blue=b
    Obj_Destroy, colors
    *self.r = Congrid(r, self.ncolors)
    *self.g = Congrid(g, self.ncolors)
    *self.b = Congrid(b, self.ncolors)
ENDIF

```



You will notice that I allow the changing of the “extra” keywords in this code. The only drawback to doing this is that *all* of the extra keywords are changed. In other words, suppose I had used two extra keywords when I initialized the object (perhaps *Keep_Aspect_Ratio* and *Divisions*). But in this method I use the extra keyword *Top*. In that case, I lose the ability to specify the *Keep_Aspect_Ratio* and *Divisions* keywords in my object. There are ways to work around this problem, of course, but it requires more programming than I care to go into in this example program. In practice, I suspect extra keywords will be rarely used.

Notice, too, that loading a new image requires that the image be loaded into all three image pointers, and that there is no way to undo this selection as a result. (We don’t have an *Undo* method written yet, in any case, but we will shortly.)

The final step is to draw the graphic, if required. Type this:

```

IF Keyword_Set(draw) THEN self->Draw
END

```

Re-compile your program, and try your new method. Type this:

```

IDL> thisObject->SetProperty, Colortable=0, /Vertical, $
      Font=0, XTitle='Earth Elevation Data', /Draw

```

The output should look similar to the illustration in Figure 122.

Notice the use of the “extra” *Font* keyword and what effect that has on the graphics output.

Get Property Methods

As important as it is to set object properties, it is equally important to know what those properties are, or to, perhaps, obtain other useful information from the object. This is usually accomplished by “get” property methods. For example, suppose you wanted to know programmatically what the current annotation color is for the object. You might write a *GetAnnotationColor* method to enquire about this color from the object. Undoubtedly, this would be a function. Add the following code to your program file somewhere in front of the *BoxImage_Define* module in the program code.

```

FUNCTION BoxImage::GetAnnotationColor
RETURN, self.annotatecolor
END

```

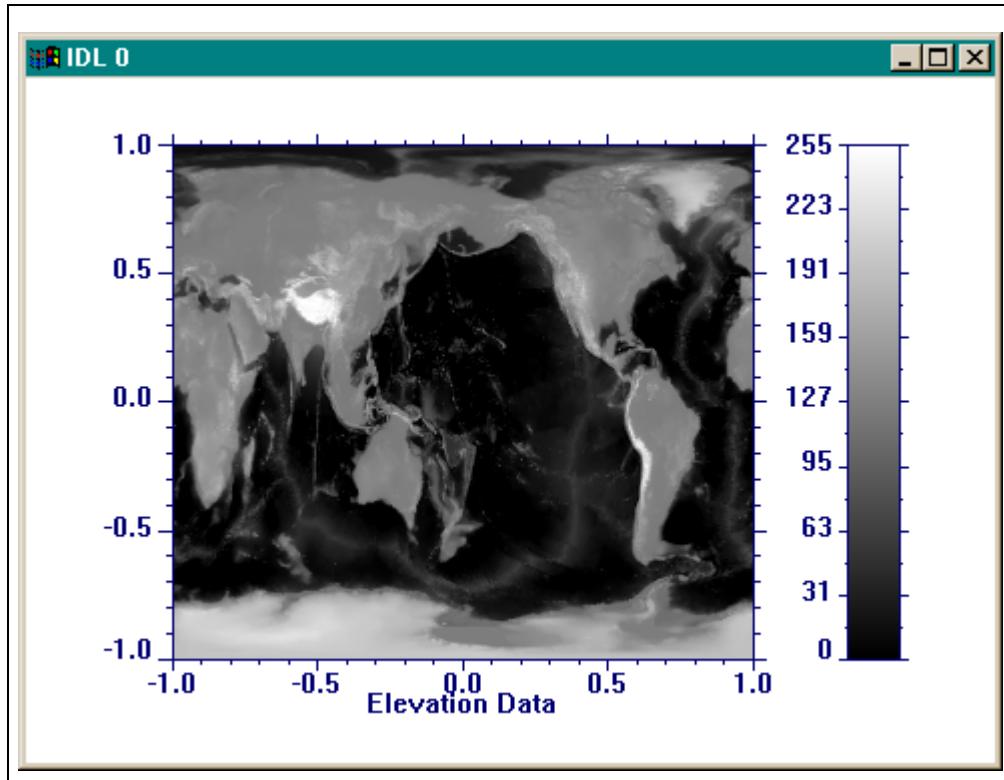


Figure 122: The *BoxImage* object after the *SetProperty* method is applied.

Recompile your program code, and try your new method:

```
IDL> annotateColor = thisObject->GetAnnotationColor()
IDL> Print, annotateColor
NAVY
```

Having a way to get and set the annotation color allows you do use programs like *PickColorName*, which you wrote in the widget programming chapters. (The program *pickcolorname.pro* is also among the programs you downloaded to use with this book.) For example, type this:

```
IDL> color = PickColorName(thisObject->GetAnnotationColor())
IDL> thisObject-> SetProperty, AnnotateColor=color, /Draw
```

You see the annotation color change to the color you selected.

While it is often useful to write get property functions, especially if you plan to use the return value of the function in an expression (as you did in the code above), it is sometimes more useful to get several properties at once via a generic *GetProperty* method. The *GetProperty* method is analogous to the *SetProperty* method you just wrote. It is a procedure and object properties are returned to the user via output keywords. For example, add this code to your program file somewhere in front of the *BoxImage_Define* module in your file.

```
PRO BoxImage::GetProperty, $
  Image = image, $
  BackColor=backcolor, $
  AnnotateColor=annotatecolor, $
  ColorTable=colortable, $
  NColors=ncolors, $
  Position=position,
```

```

    Vertical=vertical, $
    XScale=xscale, $
    XTitle=xtitle, $
    YScale=yscale, $
    YTitle=ytitle

    Catch, theError
    IF theError NE 0 THEN BEGIN
        Catch, /Cancel
        ok = Error_Message(!Error_State.Msg + ' Returning...', $
                           Traceback=1, /Error)
    RETURN
ENDIF

; Set properties if keyword is present.

IF Arg_Present(colortable) THEN colortable = self.colortable
IF Arg_Present(backcolor) THEN backcolor = self.backcolor
IF Arg_Present(annotatecolor) THEN $
    annotatecolor = self.annotatecolor
IF Arg_Present(ncolors) THEN ncolors = self.ncolors
IF Arg_Present(vertical) THEN vertical = self.vertical
IF Arg_Present(xtitle) THEN xtitle = self.xtitle
IF Arg_Present(ytitle) THEN ytitle = self.ytitle
IF Arg_Present(xscale) THEN xscale = self.xscale
IF Arg_Present(yscale) THEN yscale = self.yscale
IF Arg_Present(image) THEN image = *self.image

END

```

There are two things to notice about this code. First, I am only returning a value if the user has used the keyword and provided a valid IDL variable to receive the value. This is the purpose of using *Arg_Present*. It is overkill in this case, because normally I wouldn't care about whether the user actually asked for a value and provided a variable to hold it. I would just assign each keyword variable its appropriate object value. I do it here only because there are times when providing a value is very expensive in terms of time or computer memory, and you don't want to go to the trouble unless the user insists on it. Using *Arg_Present* is how this functionality is achieved.

The second thing to notice is that I am returning the actual image data to the user who asks for it with the *Image* keyword. What I could have returned is the pointer to the image data. In other words, I could have done this:

```
IF Arg_Present(image) THEN image = self.image
```

But this completely defeats the spirit of object encapsulation, because now someone can manipulate the object data from *outside* the object, rather than go through the object's methods. So while it is possible to do this, it is almost never a good idea. If you keep your data encapsulated inside your objects, everyone will be much happier and your programs are likely to work much better.

Re-compile your program code and test your new *GetProperty* method. Type this:

```
IDL> thisObject->GetProperty, NColors=ncolors, XTitle=xtitle
IDL> Print, 'NColors: ', ncolors
IDL> Print, 'X Title: ', xtitle
```

You can find a copy of the *BoxImage* object program with the changes made so far among the programs you downloaded to use with this book. The file should be explicitly compiled before it is used. It is named *boximage_define.2.pro*.

Creating Methods to Work with Colors in Objects

You have already seen how you can change the annotation color in the *BoxImage* object with the *GetProperty* method and the *AnnotateColor* keyword. But you can also create more elaborate color tools for the user. For example, you can incorporate the *PickColorName* program directly into an object method. For example, here is a very simple method to allow the user to select a color directly for the annotation color. Add the following code to your program file in front of the *BoxImage__Define* module in the program code.

```
PRO BoxImage::AnnotateColor, Draw=draw, _Extra=extra
thisColorName = PickColorName(self.annotatecolor, $
    Cancel=cancelled, _Extra=extra, Title='Annotation Color')
IF cancelled THEN RETURN
self.annotatecolor = thisColorName
IF Keyword_Set(draw) THEN self->Draw
END
```

Recall that the *PickColorName* program is a blocking widget program when called from the IDL command line and a modal widget program when called from within a widget program (if the *Group_Leader* keyword is used). (See “Defining a Modal Top-Level Base” on page 328 for additional information.) In this method, of course, we haven’t defined any *PickColorName* keywords (others are *Bottom* and *Title*), but we can use them if we wish, since we have defined an *_Extra* keyword to pick up extra or undefined keywords and pass them directly to the *PickColorName* program via the keyword inheritance mechanism.

In the method above, we only set the annotation color name in the object if the user didn’t cancel out of the *PickColorName* program.

Save and re-compile your program file and test this new method. Type this command:

```
IDL> thisObject->AnnotateColor, /Draw, Bottom=100
```

You should see something similar to the illustration in Figure 123 appear on your display. Select a new annotation color and see what happens to your object display.

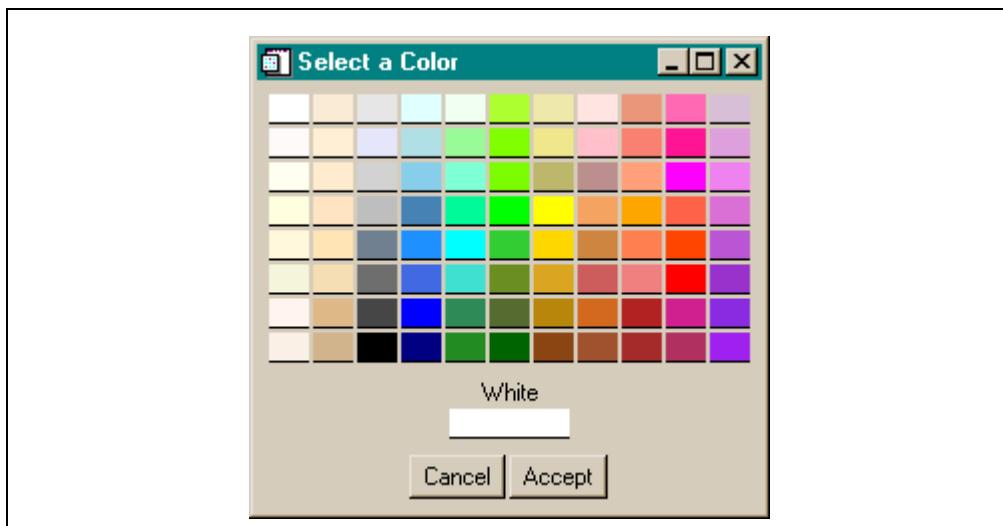


Figure 123: The *PickColorName* program called from the *AnnotateColor* method.

You can write a similar method (or modify this one if you like) to modify the background color of the object. Add this method to your program file:

```

PRO BoxImage::BackgroundColor, Draw=draw, _Extra=extra
thisColorName = PickColorName(self.backcolor, $
    Cancel=cancelled, _Extra=extra, Title='Background Color')
IF cancelled THEN RETURN
self.backcolor = thisColorName
IF Keyword_Set(draw) THEN self->Draw
END

```

Using blocking or modal keywords in object methods is fairly straightforward. But what if you want to use a non-blocking or non-modal color changing tool? For example, many people like to have a tool like *XColors* on the display so they can try a number of color tables out to see which one looks best with their image data. Is it possible to use something like this with objects?

Yes, of course, but you have to have some method of communicating between the tool that is changing the color tables and the object itself, which must respond in some way when the color table vectors change. In widget programs you sent an event to a specific widget when the color tables changed. (See “Communicating in *XColors* via Widget Events” on page 306 for additional information.)

You cannot send an event to a non-widget object, of course, but you can do something similar if the program has been written correctly. You can call an object method. For example, the *XColors* program, which you downloaded to use with this book, has been written with a *NotifyObj* keyword. The keyword accepts a structure (or array of structures, if you have more than one object to notify) that contains an object reference and the name of an object method. *XColors* will call the object method on the object reference when it changes the color table vectors. Let’s name the method *XColors* and add it to the program file in front of the *BoxImage__Define* module in the program code. It will be written like this:

```

PRO BoxImage::XColors, _Extra=extra
TVLCT, *self.r, *self.g, *self.b
struct = { XCOLORS_NOTIFYOBJ, $%
    object: self, $% ; The object reference.
    method: 'LoadColorVectors' } ; The object method.
XColors, NotifyObj=struct, NColors=self.ncolors, $%
    _Extra=extra, Title='Modify BoxImage Image Colors'
END

```

Notice that the object’s image color vectors are loaded before *XColors* is called. This is because *XColors* uses the colors in the current device color table to initialize its display. We want to make sure these colors are the colors associated with the object and not some other colors.

And notice that the object to notify is *this* object, or the *self* object. *XColors* will call the *LoadColorVectors* method, which we have yet to write. You can probably imagine what the method has to do: pull the object’s image colors out of the current color table vectors and store those in the appropriate fields of the object. The code will look like this. Add this code to your program file in front of the *BoxImage__Define* module.

```

PRO BoxImage::LoadColorVectors, _Extra=extra
TVLCT, r, g, b, /Get
*self.r = r[0:self.ncolors-1]
*self.g = g[0:self.ncolors-1]
*self.b = b[0:self.ncolors-1]
self->Draw
END

```

Notice the *_Extra* keyword that is defined for this *LoadColorVectors* method. *XColors* has the ability to pass “extra” keywords to this method that are collected on the *XColors* command line. In this case, we are not expecting any keywords, but the *_Extra* keyword must still be defined. If keywords do come into the method, they will be ignored.

Notice that the color vectors are obtained from the current color table values, and the object’s graphical display is drawn after the color table values have been stored. This puts the new colors on the display. (This is not strictly required if you are running on an 8-bit display. Can you modify the routine to reflect this fact?)

Save your program and try using the *XColors* method. Does it work the way you expect it to?

```
IDL> thisObject->XColors
```

You can find a file with the additions and modifications you have made so far among the program files you downloaded to use with this book. The file is named *boximage_define.3.pro*.

Creating Methods to Extend Object Functionality

I have to warn you that object programming is addictive. And it is especially so when it comes to writing methods that extend program functionality. The problem, simply, is that it is so easy to write object methods that extend object functionality that you never get finished writing them. Your object program just grows and grows, with each new method you write spawning five new ideas for other functionality that you can add in about 10 minutes time. It just never seems to stop. I’ve never experienced such a fecundity of ideas in regular IDL programming. But it does happen almost every time I write an object program.

We are going to restrain ourselves in the interest of instructional clarity and add the bare minimum of functionality to this object. Then, in the next section, we will see how to build ever more sophisticated objects by using objects we have already built.

The functionality I would like to add here is a bit of image processing capability, of the sort we added in the *Histo_GUI* widget program we wrote earlier. Specifically, I would like to have the ability to perform two different types of smoothing operations, two types of edge enhancement operations, and have the ability to restore the original image. In addition, I would like the ability to undo/redo a processing step that I had just applied to the image.

To keep things simple, we are going to write a *Process* method that accepts a single string command, which will be the type of image processing we desire. The five possibilities will be these: *Smooth*, *Median*, *Sobel*, *UnsharpMask*, and *Original*. Any other string will produce an error message.

Add the following code to your program file before the *BoxImage_Define* module.

```
PRO BoxImage::Process, thisProcess, Draw=draw
Catch, theError
IF theError NE 0 THEN BEGIN
    Catch, /Cancel
    ok = Error_Message(!Error_State.Msg + ' Returning...', $
        Traceback=1, /Error)
    *self.process = *self.undo
    RETURN
ENDIF
*self.undo = *self.process
```

```

CASE StrUpCase(thisProcess) OF
  'MEDIAN': *self.process = Median(*self.process, 5)
  'SMOOTH': *self.process = Smooth(*self.process, 7, $
    /Edge_Truncate)
  'SOBEL': *self.process = Sobel(*self.process)
  'UNSHARPmask': *self.process = Smooth(*self.process, 7) $-
    *self.process
  'ORIGINAL': *self.process = *self.image
ELSE: BEGIN
  msg = "Process " + StrUpCase(thisProcess) + " unknown."
  Message, msg, /NoName
END
ENDCASE

IF Keyword_Set(draw) THEN self->Draw
END

```

Notice that before the image processing is done the *undo* image is set equal to the *processed* image. This will allow us to return to the last image before the current image processing step in an *Undo* method we have yet to write. Notice, too, that we have added a new step to the standard error handling code we have used in most of these methods. The extra line restores the previous image before returning from the method. This ensures that if an error occurs no damage will have been done by executing the code in the method.

The image processing itself is done always on the *processed* image. This allows us to perform the processing steps sequentially. Of course, returning to the original image is only possible because we have the original image in the object. Note that we are forcing the command name into uppercase characters for processing in the *Case* statement.

As always when we are making changes to the object, we have the ability to draw the object immediately and see what changes have been made by means of the *Draw* keyword.

Save your program, re-compile the code and try this new *Process* method. Type these commands.

```

IDL> thisObject->Process, 'Smooth', /Draw
IDL> thisObject->Process, 'UnsharpMask', /Draw

```

Your output should look similar to the illustration in Figure 124.

Now let's add an *Undo* method. All the method has to do is switch the *undo* image with the *processed* image. The code should look like this. Add the following code to your program file before the *BoxImage__Define* module.

```

PRO BoxImage::Undo, Draw=draw
temp = *self.undo
*self.undo = *self.process
*self.process = temp
IF Keyword_Set(draw) THEN self->Draw
END

```

Save your file, re-compile the code and see if the *Undo* method works. Type this command:

```

IDL> thisObject->Undo, /Draw

```

The display output should revert back to the smoothed image. To re-apply the unsharp masking operation, simply call the *Undo* method again. Type this:

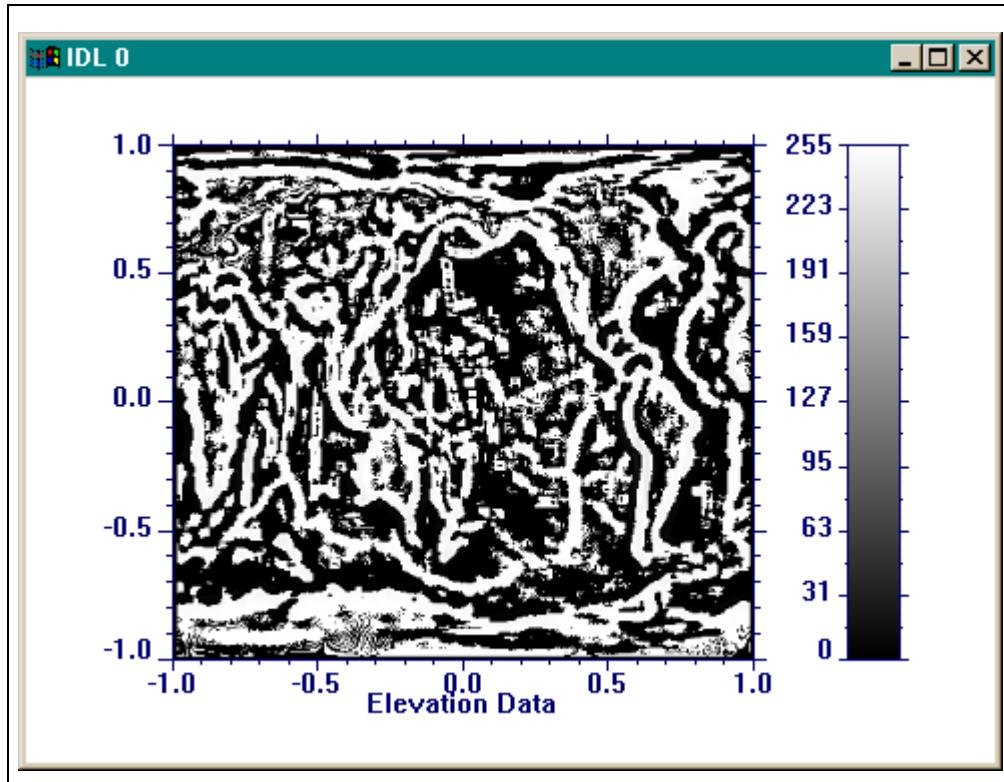


Figure 124: The *BoxImage* object after a couple of image processing steps are applied with the *Process* method.

```
IDL> thisObject->Undo, /Draw
```

You should be back to the unsharp masked image illustration in Figure 124.

You can find a copy of the *BoxImage* object program with the additions we have added so far among the programs you downloaded to use with this book. The file is named *boximage_define.pro*. The complete code is also listed in Appendix C of this book on page 399.

Object Inheritance

The strongest argument for object-oriented programming methods comes from the ability to re-use objects that have already been created to create new objects. This is known as *object inheritance*. If you design objects sensibly, with object inheritance in mind, then you can find yourself saving a great deal of work as you create new, more powerful objects out of the objects you have in your personal object library.

An object that is inherited by another object is called the *superclass* object. The object that is doing the inheriting is called the *subclass* object. (I always have to think about this a minute because I tend to get these names mixed up. It helps if I think of the object coming first as being above the object that is inheriting it, sending roots down through it.)

Subclass objects can inherit not only the encapsulated data of a superclass object, but they can inherit the superclass object's methods as well. This means that if a subclass object is not too different from a superclass object, then most of its methods will continue to work without having to make any changes. At most, you may have to create a few new methods and you may have to modify a few superclass methods. The modifi-

cation of a superclass method is generally referred to as *method overriding*. It means, in effect, that you create a method for the subclass object that has the same name as a method for the superclass object, but it works anywhere from slightly to completely differently.

Defining the Subclass Object

To see how object inheritance works, let's create a *HistoImage* subclass object by inheriting the *BoxImage* object you just created. The purpose of the *HistoImage* object will be to display an image and a color bar next to a histogram plot of the image. It will be similar to the *HistoImage* program you wrote in an earlier chapter. (See "The *HistoImage* Program" on page 240 for additional information.)

Open a new editor window and save the file as *histoimage_define.pro*. (Note, again, the two underscore characters in the name. This is critical for defining an object.) The first module in the file will be a procedure named *HistoImage_Define*. Its purpose, of course, is to define the *HistoImage* object class.

In addition to the data required in the *BoxImage* object, we will want to save data about the histogram plot we plan to add to the object. A bin size and a maximum value for the histogram plot, and a data color to draw the histogram data in are all the additional data items required. The code for the *HistoImage_Define* module will look like this:

```
PRO HistoImage__Define
    struct = { HISTOIMAGE, $
               INHERITS BoxImage, $
               binsize: 0.0, $
               max_value: 0.0, $
               datacolor: "" $
}
END
```

Note the word *INHERITS* in the second line of the structure definition. Notice there is no colon, comma, or any other punctuation between the word and the next word that follows it on the line. The *INHERITS* word adds the *BoxImage* structure definition inline at this location. In other words, it is just as if you had typed the entire structure definition (all of the structure fields) of *BoxImage* at this location in the *HistoImage* structure definition. The *INHERITS* word can go anywhere in the structure definition and there can be more than one of them. If there are more than one, then the order in which they occur becomes important.



Note that the fields in *any* structure definition in IDL must be unique. The same goes for the fields defined by an inherited structure. Every field in the structure must be unique. If you keep this in mind when you are writing objects you believe might be subclassed, then you will save yourself time and effort latter on.

While the code above is a structure definition (objects are implemented as named structures), it is more importantly an object class definition. And it is critical to realize that in an object class definition not only are the superclass object's structure fields inherited in the subclass definition, but that the superclass object's methods are also inherited.

Consider, for example, an object, *Object4*, that was subclassed from three superclass objects, like this:

```
PRO Object4__Define
    struct = { OBJECT4, $
               INHERITS Object1, $
```

```

        INHERITS Object2, $
        INHERITS Object3, $
    }
END

```

If a *SuperSize* method were called on the object like this:

```
object4->SuperSize
```

IDL would look first to see if a method *Object4::Supersize* was defined for this object. If it was not, then it would look to see if a method *Object1::Supersize* was defined, since that is the first inherited object. Failing to find such a method there, IDL would look for a method named *Object2::SuperSize*, because that was the next object inherited, etc. If *Object2* was itself a subclass of *Object5*, then IDL would search for the method *Object5::SuperSize* before it looked for *Object3::SuperSize*, since *Object5* is at a higher inherited level than *Object3*. Only if IDL could not find a *SuperSize* method in any of the inherited objects would it cause an error.

In fact, all five objects discussed here could each have its own *SuperSize* method defined for it. IDL would use the first *SuperSize* method it came to as it worked its way down the list of inherited objects. This ability to supersede a superclass method by one at a higher inherited level is called *method overriding* and is one of the features that makes object programming so powerful.

Creating Subclass Lifecycle Methods

The *HistoImage* object is only slightly different from the *BoxImage* superclass object, so it will be possible to re-use many of the *BoxImage* methods. Other methods will require method overriding, but wholesale re-writing of them won't be necessary either, since much of the functionality we need is already available. Nothing could illustrate this concept better than the two lifecycle methods, *Init* and *Cleanup*, we want to add to the *HistoImage* object.

Let's consider the *Init* method first. We have added some new data to the *HistoImage* object that is not present in the *BoxImage* object: namely the histogram bin size, the maximum value of the histogram plot, and a new data color. Like other object parameters, we would like to have the ability to define these values initially when we create the object. Normally, this is done with keywords to the *Init* method.

Add the following code to the *histogram_define.pro* program file somewhere in front of the final *Histogram_Define* module.

```

FUNCTION HistoImage::Init, $
    image, $
    Binsize=binsize, $
    DataColor=datacolor, $
    Max_Value=max_value, $
    _Extra=extra

```

Notice that in addition to the three keywords that will accept values for the three new parameters, we have defined an *image* parameter and an *_Extra* keyword. The purpose of the *_Extra* keyword is to collect any "extra" keywords used in object creation that can be sent to the *BoxImage* superclass object. In other words, rather than defining (again) all the keyword parameters that may be appropriate for the *BoxImage* object in this *HistoImage Init* method, we are going to collect them here and pass them along to the *BoxImage Init* method as a group. The *image* parameter is required, since it is a positional parameter, rather than a keyword parameter, to the *BoxImage* superclass *Init* method. If we don't collect the image data here, we have no way of automatically passing it to the superclass object.

So, the general rule is this. Any positional parameters on superclass objects should be re-declared on the function definition statement of the subclass *Init* method. Any superclass keyword parameters can be gathered using the keyword inheritance mechanism provided by the *_Extra* keyword. (Of course, you can write your objects any way you like, and ignore any “rule” you like. I sometimes re-define all the keyword parameters for the subclass object, simply because it makes the subclass object easier to document internally.)

The next step is to add error handling and a check for all the defined positional and keyword parameters. Add these lines to your program code:

```

Catch, theError
IF theError NE 0 THEN BEGIN
  Catch, /Cancel
  ok = Error_Message(!Error_State.Msg + ' Returning...', $
    Traceback=1, /Error)
  RETURN, 0
ENDIF

IF N_Elements(image) EQ 0 THEN image = LoadData(7)
ndims = Size(image, /N_Dimensions)
IF ndims NE 2 THEN $
  Message, 'Image must be 2D array.', /NoName

IF N_Elements(datacolor) EQ 0 THEN datacolor = "RED"
IF N_Elements(max_value) EQ 0 THEN max_value = 5000.0
IF N_Elements(binsize) EQ 0 THEN BEGIN
  range = Max(image) - Min(image)
  binsize = 2.0 > (range / 128.0)
ENDIF

```

You see another reason for having the image in this module: it is needed to calculate the bin size if a bin size is not supplied to the program.

The next step is to pass on the extra keywords to the *BoxImage::Init* method. You may recall I said the *Init* method, as a lifecycle method, cannot be called directly. It can only be called by IDL when an object is created. This is almost true. There is only one exception. The *Init* method of an object can be called directly, but only if it is being called from the *Init* method of a subclassed object.

Recalling that the *Init* method is a function, and that a 0 should be returned if the *Init* method fails for whatever reason, we call the *BoxImage Init* method like this:

```
IF NOT self->BoxImage::Init(image, _Extra=extra, $
  NColors=!D.Table_Size-4) THEN RETURN, 0
```

Notice how the name of the superclass object is inserted in front of the method name.

The *NColors* keyword is a *BoxImage::Init* method keyword and is used here to set the number of colors set aside for the image colors. This is required in this instance because the default number of image colors, as determined by *BoxImage*, assumes only two drawing colors. This *HistoImage* object uses three drawing colors. Without setting this keyword, the image colors would impinge on the drawing colors. Note that if the keyword is used with the *HistoImage::Init* method and is present in the *extra* structure, its value will override the value set here. This could cause a potential problem. If you thought the chance of a problem was high enough, you would probably choose to define an explicit *NColors* keyword for the *HistoImage::Init* method.

If the *BoxImage::Init* method is called successfully, most of the *self* fields have now been populated correctly with data. It remains only to populate those *self* fields that

are new in the *HistoImage* object. The final lines of code in the *Init* method look like this:

```
self.max_value = max_value  
self.datacolor = datacolor  
self.binsize = binsize  
RETURN, 1  
END
```

The purpose of the *Cleanup* method, of course, is to free pointers and other things that use memory in the object. Since we have added nothing like that in the *HistoImage* object, we can continue to use the *BoxImage::Cleanup* method. We don't have to do anything special to do so, except just not create a *Cleanup* method for this object. The inherited *BoxImage::Cleanup* method will be found and called automatically when the *HistoImage* object is destroyed.

Attaching Methods to Superclass Objects

+ Please note that if you choose *not* to create a method for a subclass object, and choose instead to use a superclass method, that you will have to live with this choice for the entire IDL session once the first object of this class is created. In other words, once you create an object in the IDL session the methods available to that object are "attached to" or "associated with" that object. You cannot change that attachment, even if you recompile the program code and create a new instance of the object. The only way you can change the attachment is by exiting IDL, or by resetting the IDL session if you are in IDL 5.3 or higher.

For example, if you were to create a *HistoImage* object now (please don't do this yet), the *BoxImage::Cleanup* method is associated with the object. If you later decide that you want a *HistoImage::Cleanup* method, you can add it to the program file, re-compile the file, create a new *HistoImage* object, etc. But you will notice that the *HistoImage::Cleanup* method will never get called. The *BoxImage::Cleanup* method is permanently attached to the *HistoImage* object in this IDL session. The only way to change this association is to exit IDL or reset the IDL session.

This restriction does not apply to new methods you want to add to the object. To add new methods, all you have to do is add them to the program code and re-compile the program code. You don't even have to re-create the object. Nor does it apply to changes you make in a method that has already been associated or attached to an object. You can make changes to the method and have the changes put into effect simply by re-compiling the program code; you do not have to re-create the object. But to override a superclass method, once you have created the subclass object in that IDL session, you must either exit or reset the IDL session.

+ Pay particular attention to this restriction because you will find no mention of it in the IDL documentation.

Overriding Superclass Methods

With the above restriction in mind, it would be a good idea now to define stubs for the superclass methods we wish to override in the *HistoImage* object. We will have to override the *Draw* method, of course, because we want to have a histogram plot in addition to the image and color bar displayed in the output window. And we have new data in the *HistoImage* object, so we will probably want to override both the *GetProperty* and *SetProperty* methods so we can access these object data values. Let's add the following lines of code to our *histoimage_define.pro* program file, somewhere in front of the *HistoImage_Define* module.

```
PRO HistoImage:: SetProperty  
END
```

```

PRO HistoImage::GetProperty
END

PRO HistoImage::Draw
END

```

Overriding the *SetProperty* method is similar to overriding the *Init* method, as we just did. We need to define keywords to set the new properties, but we can collect all the old *BoxImage* keywords with the *_Extra* keyword inheritance mechanism and pass these keywords along to the *BoxImage::SetProperty* method.

The filled out *SetProperty* method will look like this, with the additions written in bold text:

```

PRO HistoImage:: SetProperty, $
  Binsize=binsize, $
  DataColor=datacolor, $
  Max_Value=max_value, $
  _Extra=extra

  Catch, theError
  IF theError NE 0 THEN BEGIN
    Catch, /Cancel
    ok = Error_Message(!Error_State.Msg + ' Returning...', $
      Traceback=1, /Error)
    RETURN
  ENDIF

  IF N_Elements(binsize) NE 0 THEN self.binsize = binsize
  IF N_Elements(datacolor) NE 0 THEN $
    self.datacolor = datacolor
  IF N_Elements(max_value) NE 0 THEN $
    self.max_value = max_value

  self->BoxImage:: SetProperty, _Extra=extra

END

```

The *GetProperty* method is similar, but with a small twist. The *_Extra* keyword inheritance mechanism we used in the *SetProperty* method is a *pass by value* type of system. That is to say, the keywords packaged by IDL into the *extra* variable are passed to the next routine (the *BoxImage::SetProperty* method, in this case) by value and not by reference. In other words, a copy of the keyword value is passed to the routine, rather than the keyword variable itself.

This is not a problem when you are accepting input keywords, but it becomes a serious deficiency when you are trying to pass output keywords into a superclass method and obtain information back from the method. Practically speaking, this is because you can't get any information back from a *pass by value* keyword. (See "Passing Information by Reference or by Value" on page 217 for additional information.) The only thing to do, is to use the *_Ref_Extra* keyword inheritance mechanism, which passes the keywords by reference, rather than by value. (See "Using Keyword Inheritance with Output Parameters" on page 219 for additional information.)

The *GetProperty* method looks like this. Add the code in bold below to the *GetProperty* stub code you wrote earlier.

```

PRO HistoImage::GetProperty, $
  Binsize=binsize, $
  DataColor=datacolor, $
  Max_Value=max_value, $
  _Ref_Extra=extra

```

```

Catch, theError
IF theError NE 0 THEN BEGIN
    Catch, /Cancel
    ok = Error_Message(!Error_State.Msg + ' Returning...', $
        Traceback=1, /Error)
    RETURN
ENDIF

IF Arg_Present(binsize) NE 0 THEN binsize = self.binsize
IF Arg_Present(datacolor) NE 0 THEN $
    datacolor = self.datacolor
IF Arg_Present(max_value) NE 0 THEN $
    max_value = self.max_value

self->BoxImage::GetProperty, _Extra=extra
END

```

+ Note the way the *_Ref_Extra* keyword works. We have to use a *_Ref_Extra* keyword to collect the keywords in this method. But once the keywords are collected, they are passed along to the *BoxImage::GetProperty* method using the *_Extra* mechanism.

The final (and most important) method to override for the *HistoImage* object is the *Draw* method. It is the way the graphics are displayed that is the primary difference between the *BoxImage* and *HistoImage* objects. But I would like to delay writing the *Draw* method for just a couple of minutes more. I wish to talk about another property of objects: *polymorphism*. And that property is easily illustrated in the way we will write the *Draw* method. For now, let's move on to the one new method we need to write for the *HistoImage* object, the *DataColor* method.

Creating New Methods for the SubClass Object

The *BoxImage* object comes with methods to change the background color and annotation color of the graphics display. We can continue to use these methods to change these colors in the *HistoImage* object. But the *HistoImage* object has a third color, a data color, and we need a similar method to change its color.

The code will be a virtual copy of the other two color changing methods, and will look like this:

```

PRO HistoImage::DataColor, Draw=draw, _Extra=extra
thisColorName = PickColorName(self.datacolor, $
    Cancel=cancelled, _Extra=extra, Title='Data Color')
IF cancelled THEN RETURN
self.datacolor = thisColorName
IF Keyword_Set(draw) THEN self->Draw
END

```

Object Polymorphism

I think the word *polymorphism* has frightened more people away from objects than any other of the big words tossed around by the object-oriented crowd to protect their turf. Most people are quite disappointed to learn that it doesn't mean anything exotic. It simply means that a single method is able to do different things in different ways without the user of the method knowing (or caring, generally) how it happens.

For example, sending graphics output to the display, to a file, and to a printer are three different ways to display graphics output. But these different functions can be incorporated into a single *Draw* method. That is the *essence* of object polymorphism.

We will incorporate object polymorphism into the *Draw* method of the *HistoImage* object by writing it in such a way as to be able to output the graphics display to a window on the display device (or to a draw widget, for that matter), to a BMP, GIF, JPEG, PICT, PNG, TIFF, or PostScript file, and directly to the default printer.

Replace the *HistoImage*::*Draw* method code stub you wrote earlier with this code.

```
PRO HistoImage::Draw, $
    Font=font, $
    BMP=bmp, $
    GIF=gif, $
    JPEG=jpeg, $
    PICT=pict, $
    PNG=png, $
    TIFF=Tiff, $
    PostScript=postscript, $
    PS=ps, $
    Printer=printer, $
    _Extra=extra
```

The *Font* keyword will allow the user to specify a different type of font (e.g., hardware or true-type) for output display. The other keywords allow the user to select a different kind of file or printer output. If these keywords are not set, the output is sent to the current graphics device, as normal. Notice that both a *PostScript* and a *PS* keyword are defined. We will write the code so that the user can use either of these two keywords to select PostScript file output. The *_Extra* keyword will pick up other keywords that we might want to use to configure programs like *PSCconfig* or *PSWindow*, or the Z graphics buffer, which we will be using in the program.

Next, add the error handling code. This is standard operating procedure by now.

```
Catch, theError
IF theError NE 0 THEN BEGIN
    Catch, /Cancel
    ok = Error_Message(!Error_State.Msg + ' Returning...', $
        Traceback=1, /Error)
    RETURN
ENDIF
```

The next step is to check the value of the keywords. We will use the file output display keywords to set the value of an *output* variable. This will mean that only one of these keywords can be active at any one time. The code will look like this:

```
IF N_Elements(font) EQ 0 THEN font = !P.Font
output = ""
IF Keyword_Set(bmp) THEN output = 'BMP'
IF Keyword_Set(gif) THEN output = 'GIF'
IF Keyword_Set(jpeg) THEN output = 'JPEG'
IF Keyword_Set(pict) THEN output = 'PICT'
IF Keyword_Set(png) THEN output = 'PNG'
IF Keyword_Set(tiff) THEN output = 'TIFF'
IF Keyword_Set(postscript) THEN output = 'PS'
IF Keyword_Set(ps) THEN output = 'PS'
IF Keyword_Set(printer) THEN output = 'PRINTER'
```

If the *output* variable is not a null string, then we are going to switch to another graphics device to display program output. We should save the current device name so we can restore it later. Type this command:

```
IF output NE "" THEN thisDevice = !D.Name
```

The next step is to perform different setup operations, depending upon the value of the *output* variable. The *Case* statement outline allowing us to do this will look like this:

```
CASE output OF
  "": BEGIN
    END
  "PS": BEGIN
    END
  "PRINTER": BEGIN
    END
  ELSE: BEGIN
    END
ENDCASE
```

Consider first the case when the *output* variable is a null string. This indicates that the object's graphics output will be sent to the display. The only preparation required in this case is to load the drawing and image colors. The code (in bold) looks like this:

```
"": BEGIN
  annotateColor = GetColor(self.annotateColor, $
    !D.Table_Size-2)
  backColor = GetColor(self.backColor, $
    !D.Table_Size-3)
  dataColor = GetColor(self.dataColor, $
    !D.Table_Size-4)
  TVLCT, *self.r, *self.g, *self.b
END
```

If the *output* variable is set to indicate PostScript output, then we should configure the PostScript device according to the users wishes. We can do this with the *PSCconfig* program you downloaded to use with this book. Moreover, we will want to make sure we are selecting proper colors for PostScript output. We will select colors here that will be appropriate for either black and white, or color PostScript output. (I am making arbitrary decisions about these things, based on the kinds of printers I have in my office and am likely to print the output on. Your decisions should be based on your own local situation. You may not change the objects drawing colors at all, for example, if you always send output to a color PostScript printer.) The code will look like this:

```
"PS": BEGIN
  keywords = PSCconfig(Color=1, _Extra=extra, $
    Filename='histoimage.ps', Cancel=cancelled)
  IF cancelled THEN RETURN ELSE keywords.color = 1
  Set_Plot, 'PS'
  Device, _Extra=keywords
  annotateColor = GetColor('Navy', !D.Table_Size-2)
  backColor = GetColor('White', !D.Table_Size-3)
  dataColor = GetColor('Black', !D.Table_Size-4)
  TVLCT, *self.r, *self.g, *self.b
END
```

Notice that I set the *Color* keyword in *PSConfig* to start with. And I make sure it is set again before I pass the keywords to the PostScript device. This is because I will not be able to get accurate PostScript output of my graphical display unless this keyword is set. Once in the PostScript device, I load my drawing and image colors.

The next case deals with the *Printer* device. Getting reliable output to the *Printer* device is always tricky. There are so many different kinds of printers, with so many printer drivers, etc. Sometimes I think it is remarkable that we can get any output, period.

One of the interesting things I have learned about the *Printer* device is that I almost always have better luck loading colors outside the device and copying the colors to the device than I do loading the colors once I am inside the device. I don't know why this is. Nothing in the IDL documentation suggests this should be an issue, but whenever I have problems getting output to show up on the printed page it involves some kind of a color loading problem. (And there are known bugs in the *Printer* device with respect to loading color tables. See "Loading Colors in the Printer Device" on page 204 for additional information.)

So, with this in mind, I will write the *Printer* device part of the code like this:

```
"PRINTER": BEGIN
    ok = Dialog_PrinterSetup()
    IF NOT ok THEN RETURN

    annotateColor = GetColor('Black', !D.Table_Size-2)
    backColor = GetColor('Charcoal', !D.Table_Size-3)
    dataColor = GetColor('Black', !D.Table_Size-4)
    TVLCT, *self.r, *self.g, *self.b

    keywords = PSWindow(/Printer, /Landscape, $
        Fudge=0.25, _Extra=extra)
    Set_Plot, 'PRINTER', /Copy
    Device, Landscape=1
    Device, _Extra=extra

    thisThickness = !P.Thick
    thisFont = !P.Font
    font = 1
    !P.Thick = 2
END
```

There are several things to notice in this code. First, I allow the user access to the *Printer* device dialog. This will allow the user to select a different networked printer, etc. If the user cancels out of this dialog, I will exit this method. Second, I load the drawing and image colors before I make the *Printer* device the current graphics device. Third, I am going to make a window on the *Printer* device with the current window's aspect ratio. I am arbitrarily selecting landscape output and I am not offering the user a choice about this. (Such choices could easily be built into the object itself, of course.)

Notice that I set the *Landscape* keyword on the *Device* command independently of setting the other keywords. I've found this is the only reliable way to make sure the offsets and sizes returned from the *PSWindow* function (which you downloaded to use with this book) are recognized by the *Printer* device. Notice, too, that I have set a *Fudge* factor of 0.25 inches. This corresponds to the printable area on my printer where the offset point is calculated. Your printer may use the corner of the actual page for calculating offsets, or you may need other fudge factors than these. You will have to determine this empirically from your printer output. (See "Positioning Graphics with the Printer Device" on page 202 for additional information.)

Fourth, I change the default font to true-type output and I set the thickness of all plotted lines to double thickness. I do this because single lines on my 600 dpi printer are usually too thin to show up well.

The final case involves any other kind of file output. In such a situation, rather than drawing the graphics output into a window, I am going to draw the output into the Z-graphics buffer. This way I will be able to take a snapshot or screen dump of the Z-buffer and make the proper type of output file with the snapshot. The code for this case will look like this:

```

ELSE: BEGIN

    ncolors = !D.Table_Size
    Set_Plot, 'Z'
    Device, Set_Resolution=[500, 500], $
        Set_Colors=ncolors, _Extra=extra
    Erase
    annotateColor = GetColor(self.annotateColor, $
        ncolors-2)
    backColor = GetColor(self.backColor, ncolors-3)
    dataColor = GetColor(self.dataColor, ncolors-4)
    TVLCT, *self.r, *self.g, *self.b

END

ENDCASE

```

Notice that this kind of file output will always result in a 500 by 500 image. Also, I will be using the number of colors available in the current graphics device (or 256, whichever is smaller), rather than the number of colors available in the Z-graphics buffer. This will result in graphics output that is identical (essentially) to the output I see on the display, no matter what depth of graphics display I have. The *Erase* command erases the display, so there are no left-over graphics in the Z-buffer.

The next step in creating the *Draw* method is to calculate positions for the image, color bar, and histogram plot from the overall position of the output in the object. This will be different if we have a vertical color bar as opposed to a horizontal one. The code looks like this:

```

IF self.vertical THEN BEGIN
    p = self.position
    length = p[2] - p[0]
    imgpos = [p[0], p[1], (p[0] + (0.75*length)), $ 
        p[3]-(length*0.350)]
    cbpos = [p[2]-0.05, p[1], p[2], p[3]-(length*0.35)]
    hpos = [p[0], imgpos[3]+0.1, p[2], p[3]]
ENDIF ELSE BEGIN
    p = self.position
    height = p[3] - p[1]
    imgpos = [p[0], p[1], p[2], p[1]+ 0.4*height]
    cbpos = [p[0], imgpos[3]+height*0.1, p[2], $ 
        imgpos[3]+height*0.15]
    hpos = [p[0], cbpos[3]+height*0.125, p[2], p[3]]
ENDELSE

```

Next, I would like the character size in the graphics output to vary depending upon the size of the output window. I can use the *Str_Size* program you downloaded to use with this book to calculate a character size for me. The only time I don't want a variable string size, is when I am sending output to the *Printer* device. The code will look like this:

```
IF output EQ 'PRINTER' THEN thisCharsize = 1.25 ELSE $
    thisCharsize = Str_Size('A Sample String', 0.20)
```

Next, I need to calculate the histogram data. I will do this on the processed image, rather than the original image data, since I want the histogram to reflect the image that is shown in the display. The code looks like this:

```
histdata = Histogram(*self.process, Binsize=self.binsize, $
    Max=Max(*self.process), Min=Min(*self.process))
```

The next step is to fudge the actual histogram data to draw the bins correctly. (See “Drawing the Histogram Plot” on page 248 for additional information.) The code looks like this:

```
npts = N_Elements(histdata)
halfbinsize = self.binsize / 2.0
bins = Findgen(N_Elements(histdata)) * $
    self.binsize + Min(*self.process)
binsToPlot = [bins[0], bins + halfbinsize, $
    bins[npts-1] + self.binsize]
histdataToPlot = [histdata[0], histdata, histdata[npts-1]]
xrange = [Min(binsToPlot), Max(binsToPlot)]
```

Finally, we are ready to draw the histogram plot and the bins. The code looks like this:

```
Plot, binsToPlot, histdataToPlot, $
    Background=backcolor, $
    Charsize=thisCharsize, $
    Color=annotateColor, $
    Font=font, $
    Max_Value=self.max_value, $
    NoData=1, $
    Position=hpos, $
    Title='Image Histogram', $
    XRange=xrange, $
    XStyle=1, $
    XTickformat='(I6)', $
    XTitle='Image Value', $
    YMinor=1, $
    YRange=[0,self.max_value], $
    YStyle=1, $
    YTickformat='(I6)', $
    YTitle='Pixel Density', $
    _Extra=*self.extra

OPlot, binsToPlot, histdataToPlot, PSym=10, Color=dataColor
FOR j=1L,N_Elements(bins)-2 DO BEGIN
    Plots, [bins[j], bins[j]], [!Y.CRange[0], histdata[j] $
        < self.max_value], Color=dataColor, _Extra=*self.extra
ENDFOR
```

Display of the image and color bar can come from the *BoxImage::Draw* method. The code looks like this:

```
TVImage, BytScl(*self.process, Top=self.ncolors-1), $
    Position=imgpos, _Extra=*self.extra
Plot, self.xscale, self.yscale, XStyle=1, YStyle=1, $
    XTitle=self.xtitle, YTitle=self.ytitle, $
    Color=annotateColor, Position=imgpos, /NoErase, $
```

```

    /NoData, Ticklen=-0.025, _Extra=*self.extra, $
    CharSize=thisCharSize, Font=font
    Colorbar, Range=[Min(*self.process), Max(*self.process)], $
    Divisions=8, _Extra=*self.extra, Color=annotateColor, $
    Position=cbpos, Ticklen=-0.2, Vertical=self.vertical, $
    NColors=self.ncolors, CharSize=thisCharSize, Font=font

```

The only thing that remains to be done is to clean up based on the value of the *output* variable. This involves closing files, writing file output, and making sure we restore any system variables or graphics output devices, if they have changed. We can use the *TVRead* program you downloaded to use with this book to take a screen dump of the Z-graphics buffer and write the appropriate output file. The code looks like this:

```

CASE output OF
    "BMP": image = TVRead(/BMP, Filename='histoimage')
    "GIF": image = TVRead(/GIF, Filename='histoimage')
    "JPEG": image = TVRead(/JPEG, Filename='histoimage')
    "PNG": image = TVRead(/PNG, Filename='histoimage')
    "PICT": image = TVRead(/PICT, Filename='histoimage')
    "TIFF": image = TVRead(/TIFF, Filename='histoimage')
    "PS": Device, Close_File=1
    "PRINTER": BEGIN
        Device, Close_Document=1
        !P.Thick = thisThickness
        !P.Font - thisFont
    END
    ELSE:
ENDCASE
IF output NE "" THEN Set_Plot, thisDevice
END

```

Testing the *HistoImage* Object

You are finally ready to test the *HistoImage* object program. Save your program file, compile it, fix any errors that occur, re-compile, etc. (You can find a complete copy of the program in the file *histoimage_define.pro* that you downloaded to use with this book. A complete program listing is also available in Appendix C on page 423). Type these commands:

```

IDL> image = LoadData(7)
IDL> thisObject = Obj_New('HistoImage', image, /Vertical)
IDL> thisObject->Draw

```

Your output should look similar to the illustration in Figure 125.

Test some of the object's methods. Type this command:

```

IDL> thisObject->SetProperty, Colortable=3, Vertical=0, $
    BackColor='gray', AnnotateColor='yellow', $
    DataColor='beige', /Draw

```

Can you make a JPEG file from this output? Type this command to exercise the object's polymorphism:

```

IDL> thisObject->Draw, /JPEG

```

How about sending the output to the default printer?

```

IDL> thisObject->Draw, /Printer

```

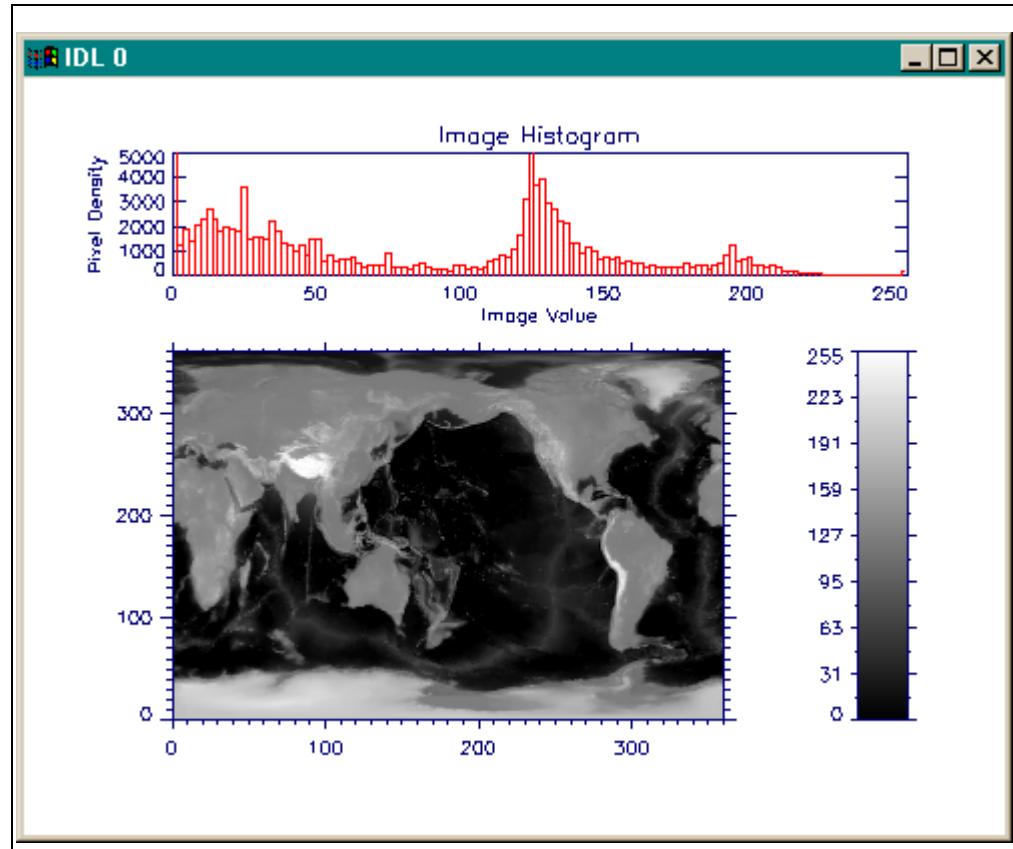


Figure 125: The graphics display of the HistoImage object Draw method.

When you are finished testing the object, be sure to destroy it, like this:

```
IDL> Obj_Destroy, thisObject
```

Appendix A

◆ Discovering the Possibilities ◆◆◆



Appendix A: Widget Event Structures

Event Structure Definition

Event structures contain information about specific widget events. Each event generates its own, specific event structure, which is dispatched to an event handler module in a widget program. Event structures are either named or anonymous IDL structure variables. What makes a structure an *event structure*, as opposed to some other kind of structure, is the presence of three fields, *ID*, *Top*, and *Handler*, which are defined as long integers. Here is a list of the event structures created and returned by widgets supplied with the IDL distribution.

Common Field Definitions

The *ID* field is the unique widget identifier of the widget that caused the event. It is always a long integer.

The widget that caused the event is part of a hierarchy of widgets. The *Top* field is the unique widget identifier of the widget that is at the top of that particular widget hierarchy. It is always a long integer.

The event structure created by the event will be sent to an event handler. Every event handler is associated with or connected to a widget. The *Handler* field is the unique widget identifier of the widget that is associated with event handler for this particular event. It is always a long integer.

Details about these common event structure fields and how they are defined can be found in “Common Fields in Event Structures” on page 277.

Basic Widget Event Structures

Base Widget Event Structure

```
{ WIDGET_BASE, ID:0L, Top:0L, Handler:0L, X:0, Y:0 }
```

A base widget only generates events if it is a top-level base and it is resized by the user. The keyword *TLB_Size_Events* must be explicitly set to generate events. The *X* and *Y* fields are the size of the top-level base in pixels. The size of the base does *not* include any window frame or menu bar attached to the top-level base by the window manager.

Button Widget Event Structure

```
{ WIDGET_BUTTON, ID:0L, Top:0L, Handler:0L, Select:0 }
```

The *Select* field is set to one if the button was set, and to zero if the button was released. Normal buttons do not generate events when released, so *Select* will always be one. However, toggle buttons return separate events for the set and release actions.

Draw Widget Event Structure

```
{ WIDGET_DRAW, ID:0L, Top:0L, Handler:0L, Type: 0, X:0, Y:0,
    Press:0B, Release:0B, Clicks:0, Modifiers:0 }
```

The *Type* field tells what kind of event this is. The possibilities are: *button down* (0), *button up* (1), *motion* (2), *viewport scroll* (3), and *expose* (4). All possible events must be explicitly set with keywords or they don't occur.

The *X* and *Y* fields give the device or pixel coordinates at which the event occurred, measured from the lower-left corner of the drawing area. *Press* and *Release* are bitmasks in which the value of the bits set represents which button was pressed or released. A value of 1 indicated the left mouse button. A value of 2 represents the middle mouse button. And a value of 4 represents the right mouse button. If the event is a motion event, both *Press* and *Release* are zero.

The *Clicks* field returns a 1 if the time interval between button-press events is greater than the time interval for a double-click event for the computer itself. If the time interval between two button-press events is less than the time interval for a double-click event for the computer, the *Clicks* field returns 2. Time intervals are computer-specific and sometimes configurable by the user. See your computer documentation for details.

The *Modifiers* field was added in IDL 5.3. It is a bit mask that allows you to obtain information about *Shift*, *Control*, *Caps Lock*, and *Alt* modify keys, usually in conjunction with *Press* and *Release* events. See the *Widget_Draw* documentation for details.

Droplist Widget Event Structure

```
{ WIDGET_DROPLIST, ID:0L, Top:0L, Handler:0L, Index:0L }
```

The *Index* field returns the index of the selected item. This can be used to index the array of names originally used to set the widget's value. (The array of names may have to be stored in the widget's user value or elsewhere.)

Label Widget Event Structure

Label widgets do not generate events on their own. A timer event (see below) may be set on a label widget, however.

List Widget Event Structure

```
{ WIDGET_LIST, ID:0L, Top:0L, Handler:0L, Index:0L,
    Clicks:0L }
```

The *Index* field 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. The *Clicks* field returns either one or two, depending upon how the list item was selected. If the list item is double-clicked, *Clicks* is set to two. Note that you get *both* the single click and double click events in your event handler. Be sure you plan accordingly.

Slider Widget Event Structure

```
{ WIDGET_SLIDER, ID:0L, Top:0L, Handler:0L, Value:0L,
  Drag:0 }
```

The *Value* field returns the new value of the slider. The *Drag* field returns a one 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 drag events when the *Drag* keyword is set, and then *only* on UNIX platforms.

Table Widget Event Structure

Character Insertion Event

```
{ WIDGET_TABLE_CH, ID:0L, Top:0L, Handler:0L, Type:0,
  Offset:0L, Ch:0B, X:0L, Y:0L }
```

The *Offset* field is the zero-based insertion position that will result *after* the character is inserted. The *Ch* field is the ASCII value of the character. The *X* and *Y* fields give the zero-based address of the cell within the table.

String Insertion Event

```
{ WIDGET_TABLE_STR, ID:0L, Top:0L, Handler:0L, Type:1,
  Offset:0L, Str:' ', X:0L, Y:0L }
```

The *Str* field is the character string to be inserted.

Delete String Event

```
{ WIDGET_TABLE_DEL, ID:0L, Top:0L, Handler:0L, Type:2,
  Offset:0L, Length:0L, X:0L, Y:0L }
```

The *Offset* field is the zero-based character position of the first character deleted. It is also the insertion position that will result when the next character is inserted. The *Length* field gives the number of characters involved.

Text Selection Event

```
{ WIDGET_TABLE_TEXT_SEL, ID:0L, Top:0L, Handler:0L, Type:3,
  Offset:0L, Length:0L, X:0L, Y:0L }
```

The event announces a change in the insertion point. The *Offset* field is the zero-based character position of the first character to be selected. The *Length* field 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*.

Cell Selection Event

```
{ 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 event announces a change in the currently selected cells. 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 de-selected (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 that this means that two *WIDGET_TABLE_CELL_SEL* events are generated when an existing selection is changed to a new selection. Be sure your code differentiates between select and deselect events.

Row Height Changed Event

```
{ WIDGET_TABLE_ROW_HEIGHT, ID:0L, Top:0L, Handler:0L,  
    Type:6, Row:0L, Height:0L }
```

The event announces that the height of the given row has been changed by the user. The *Row* field contains the zero-based row number, and the *Height* field contains the new height.

Column Width Changed Event

```
{ WIDGET_TABLE_COLUMN_WIDTH, ID:0L, Top:0L, Handler:0L,  
    Type:7, Column:0L, Width:0L }
```

The event announces that the width of the given column has been changed by the user. The *Column* field contains the zero-based column number, and the *Width* field contains the new width.

Invalid Data Event

```
{ WIDGET_TABLE_INVALID_ENTRY, ID:0L, Top:0L, Handler:0L,  
    Type:8, Str:' ', X:0L, Y:0L }
```

When this event is generated, the cell's data is left unchanged. The invalid contents entered by the user is given as a text string in the *Str* field. The cell location is given by the *X* and *Y* fields.

Text Widget Event Structure

Character Insertion Event

```
{ WIDGET_TEXT_CH, ID:0L, Top:0L, Handler:0L, Type:0,  
    Offset:0L, Ch:0B }
```

The *Offset* field is the zero-based insertion position that will result *after* the character is inserted. The *Ch* field is the ASCII value of the character.

String Insertion Event

```
{ WIDGET_TEXT_STR, ID:0L, Top:0L, Handler:0L, Type:1,  
    Offset:0L, Str:' ' }
```

The *Str* field is the character string to be inserted.

Delete String Event

```
{ WIDGET_TEXT_DEL, ID:0L, Top:0L, Handler:0L, Type:2,  
    Offset:0L, Length:0L }
```

The *Offset* field is the zero-based character position of the first character deleted. It is also the insertion position that will result when the next character is inserted. The *Length* field gives the number of characters involved.

Text Selection Event

```
{ WIDGET_TEXT_SEL, ID:0L, Top:0L, Handler:0L, Type:3,  
    Offset:0L, Length:0L }
```

The event announces a change in the insertion point. The *Offset* field is the zero-based character position of the first character to be selected. The *Length* field 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*.

Compound Widget Event Structures

CW_Animate Event Structure

```
{ ID:0L, Top:0L, Handler:0L, Action:'' }
```

The only string allowed in the *Action* field is the string “DONE”.

CW_Arcbal Event Structure

```
{ ID:0L, Top:0L, Handler:0L, Value:FltArr(3,3) }
```

The *Value* field contains a new rotation matrix.

CW_BGroup Event Structure

```
{ ID:0L, Top:0L, Handler:0L, Select:0, Value:* }
```

The *Select* field is passed through from the underlying button event. The *Value* field is either the *Index*, *ID*, *Name*, or *Button_UValue* of the button, depending on how the widget was created.

CW_Clr_Index Event Structure

```
{ CW_COLOR_INDEX, ID:0L, Top:0L, Handler:0L, Value:0 }
```

The *Value* field is the color index selected.

CW_Color_Set Event Structure

```
{ COLORSEL_EVENT, ID:0L, Top:0L, Handler:0L, Value:0 }
```

The *Value* field is the color index selected.

CW_DefROI Event Structure

This compound widget has an internal event handler and does not generate an external event structure.

CW_Field Event Structure

```
{ ID:0L, Top:0L, Handler:0L, Value:'', Type:0, Update:0 }
```

The *Value* field is the value of the field’s text widget. The *Type* field specifies the type of data contained in the field and can be: 0 (string), 1 (floating-point), 2 (integer), or 3 (long integer). The *Update* field contains a zero if the field has not been altered or a one if it has.

CW_Form Event Structure

```
{ ID:0L, Top:0L, Handler:0L, Tag:'', Value:0, Quit:0 }
```

The *Tag* field contains the tag name of the field that changed. The *Value* field contains the new value of the changed field. The *Quit* field contains a zero if the quit flag is not set, or one if it is set.

CW_FSlider Event Structure

```
{ ID:0L, Top:0L, Handler:0L, Value:0.0, Drag:0 }
```

The *Value* field is the value of the slider. The *Drag* field indicates (with a one) that the events are updated continuously (as the user drags), or only when the user releases the slider (with a zero).

CW_Orient Event Structure

This compound widget returns various types of event structures when the transformation matrix is changed by its various internal widgets. Basically, the widget passes event structures on directly from its internal widgets after first changing the *ID* field to point to the top-level base of this compound widget. Most event handlers will do well to ignore any events coming from this widget, since the *!P.T* system variable is updated automatically anyway.

CW_PDMenu Event Structure

```
{ ID:0L, Top:0L, Handler:0L, Value:* }
```

The *Value* field is either the *Index*, *ID*, *Name*, or *Full_Name* of the button, depending on how the widget was created.

CW_RGBSlider Event Structure

```
{ ID:0L, Top:0L, Handler:0L, R:0B, G:0B, B:0B }
```

The *R*, *G*, and *B* fields contain the red, green, and blue value of the selected color triple, respectively.

CW_Zoom Event Structure

```
{ ZOOM_EVENT, ID:0L, Top:0L, Handler:0L, XSize:0L, YSize:0L,
    X0:0L, Y0:0L, X1:0L, Y1:0L }
```

The *XSize* and *YSize* fields contain the dimensions of the zoomed image. The *X0*, *Y0*, *X1*, and *Y1* fields contain the coordinates of the lower-left and upper-right corners of the original image, respectively.

FSC_InputField Event Structure

```
{ FSC_FIELD, ID:0L, Top:0L, Handler:0L,
    ObjRef:Obj_New(), Value:Ptr_New(), Type:'' }
```

The compound widget FSC_InputField is written as an object and is controlled by object methods. Thus, the *ObjRef* field returns the object reference of the compound widget. This allows you to easily control various aspects of how the compound widget works. The *Value* field is a pointer to a value. This is guaranteed to be a valid pointer, but it may point to an undefined variable. You should test the result before using it in an expression. The *Type* field tells you the type of variable in the value field. Possible values are "INT", "LONG", "FLOAT", "DOUBLE", or "STRING".

Widget Program Event Structures

XColors Event Structure

```
{ XCOLORS_LOAD, ID:0L, Top:0L, Handler:0L,
    R:BytArr(!D.Table_Size), G:BytArr(!D.Table_Size),
    B:BytArr(!D.Table_Size, Index:0, Name:'' ) }
```

XColors sends a widget event to widgets identified with the *NotifyID* keyword. The *R*, *G*, and *B* fields contain the current red, green, and blue color table vectors, respectively. The *Index* field will be set to a -1 or to the index number of the currently loaded color table after a color table has been loaded. The *Name* field contains the name of the selected color table.

ReadImage Event Structure

```
{ READIMAGE_EVENT, ID:0L, Top:0L, Handler:0L, $  
    XSize:0, YSize:0, Filename:'' }
```

ReadImage sends a widget event to widgets identified with the *NotifyIDs* parameter. The *XSize* field contains the X size of the image file. The *YSize* field contains the Y size of the image file. The *Filename* field contains the name of the file.

Other Widget Event Structures

There are several other types of events that can be generated by widgets. They are listed here.

Keyboard Focus Events

```
{ WIDGET_KBRD_FOCUS, ID:0L, Top:0L, Handler:0L, Enter:0 }
```

Some widgets, such as the text widget, can have keyboard focus events turned on with the *KBRD_FOCUS_EVENTS* keyword. These widgets will generate events when the widget gains keyboard focus (the *Enter* field is set to one), or loses keyboard focus (the *Enter* field is set to zero).

Kill Widget Request Events

```
{ WIDGET_KILL_REQUEST, ID:0L, Top:0L, Handler:0L }
```

Top-level bases that have the *TLB_Kill_Request_Events* keyword set will receive this event structure if the widget is killed by the window manager (i.e., the user used the mouse to kill the widget rather than a *Quit* button). It is often easier to use a *Cleanup* procedure to handle any killed widget events, whether they come from using the *Quit* button or from the user killing the widget with the window manager.

Widget Timer Events

```
{ WIDGET_TIMER, ID:0L, Top:0L, Handler:0L }
```

Event handlers can do whatever they like when a widget timer event is received. The *ID* field is set to the identifier of the widget for which the timer is set.

Widget Tracking Events

```
{ WIDGET_TRACKING, ID:0L, Top:0L, Handler:0L, Enter:0 }
```

Tracking events are generated each time the mouse pointer passes into the boundary of a widget (the *Enter* field is set to one) or out of the boundary of a widget (the *Enter* field is set to zero). The *Tracking_Events* keyword must be set to enable widget tracking for a specific widget. Note that tracking events do not work well for IDL programs running on Windows machines.

Appendix B

♦ Discovering the Possibilities ♦♦♦



Appendix B: Data File Descriptions

Here is a descriptive list of all the data files used with this book. The files can be found in the IDL distribution in the *!DIR/examples/data* directory. The files can also be copied from their location in the IDL distribution to a specified directory. See “Copying the Data Files” on page 5 for additional information.

Name	Description	Type	X	Y	Z
abnorm.dat	Gated Blood Pool	Byte	64	64	15
cereb.dat	X Ray of Brain	Byte	512	512	1
convec.dat	Earth Mantle Convection	Byte	248	248	1
ctscan.dat	Cat Scan of Thorasic Cavity	Byte	256	256	1
galaxy.dat	Image of Galaxy	Byte	256	256	1
head.dat	MRI Slices of Human Head	Byte	80	100	57
hurric.dat	Hurricane Gilbert	Byte	440	340	1
image24.dat	Colored World Elevation Data	Byte	3	360	360
jet.dat	Hydrodynamic Simulation	Byte	81	40	101
m51.dat	Whirlpool M51 Galaxy	Byte	340	440	1
people.dat	Founders of Research Systems	Byte	192	192	2
nyny.dat	Arial Photo of New York City	Byte	768	512	1
worldelv.dat	World Elevation Data	Byte	360	360	1

Table 16: *These are the data files that should be downloaded or copied for use with this book. Use the CopyData program to copy the data files from the IDL distribution to your local directory.*

Appendix C

◆ Discovering the Possibilities ◆◆◆



Appendix C: IDL Program Code

IDL Example Programs

This appendix contains the IDL source code for selected programs used in the book. The annotated programs are listed here for reference and so they can be read and studied more easily.

BoxImage_Define Object Program

This is an example program that illustrates how to create a display object in IDL. Information about the program can be found in “Creating Graphics Display Objects” on page 349. The file can be downloaded from the *coyote* anonymous ftp site. The URL is:

ftp://ftp.dfanning.com/pub/dfanning/outgoing/coyote2nd/boximage_define.pro

```
PRO BoxImage::Undo, Draw=draw
; This method reverses the last processing step.
temp = *self.undo
*self.undo = *self.process
*self.process = temp
; Redraw the image if needed.
IF Keyword_Set(draw) THEN self->Draw
END
```

```
PRO BoxImage::Process, thisProcess, Draw=draw
; Error handling.
Catch, theError
IF theError NE 0 THEN BEGIN
  Catch, /Cancel
  ok = Error_Message(!Error_State.Msg + ' Returning...', $
    Traceback=1, /Error)
  *self.process = *self.undo
  RETURN
ENDIF
; Set up undo image.
```

Appendix C: IDL Program Code

```
*self.undo = *self.process
; Apply the process.

CASE StrUpCase(thisProcess) OF
  'MEDIAN': *self.process = Median(*self.process, 5)
  'SMOOTH': *self.process = Smooth(*self.process, 7, /Edge_Truncate)
  'SOBEL': *self.process = Sobel(*self.process)
  'UNSHARPmask': *self.process = Smooth(*self.process, 7) - *self.process
  'ORIGINAL': *self.process = *self.image
ELSE: BEGIN
  msg = "Process " + StrUpCase(thisProcess) + " unknown."
  Message, msg, /NoName
END
ENDCASE

IF Keyword_Set(draw) THEN self->Draw
END
```

```
PRO BoxImage::LoadColorVectors
; Get the current color vectors.

TVLCT, r, g, b, /Get
; Pull out the image colors.

*self.r = r[0:self.ncolors-1]
*self.g = g[0:self.ncolors-1]
*self.b = b[0:self.ncolors-1]

; Redraw the image.

self->Draw
END
```

```
PRO BoxImage::XColors, _Extra=extra
; Load the current image colors.

TVLCT, *self.r, *self.g, *self.b
; Call XCOLORS and notify this object if colors change.

struct = { XCOLORS_NOTIFYOBJ, $
  object:self, $ ; The object reference.
  method:'LoadColorVectors' } ; The object method to call.

XColors, NotifyObj=struct, NColors=self.ncolors, _Extra=extra, $
  Title='Modify BoxImage Image Colors'
END
```

```
PRO BoxImage::BackgroundColor, Draw=draw, _Extra=extra
; This method changes the background color.

thisColorName = PickColorName(self.backcolor, Cancel=cancelled, $
  _Extra=extra, Title='Background Color')
IF cancelled THEN RETURN

self.backcolor = thisColorName
; Redraw the image if needed.
```

```
IF Keyword_Set(draw) THEN self->Draw
END
```

```
PRO BoxImage::AnnotateColor, Draw=draw, _Extra=extra
; This method changes the annotation color.
thisColorName = PickColorName(self.annotatecolor, Cancel=cancelled, $
    _Extra=extra, Title='Annotation Color')
IF cancelled THEN RETURN
self.annotatecolor = thisColorName
; Redraw the image if needed.
IF Keyword_Set(draw) THEN self->Draw
END
```

```
PRO BoxImage::GetProperty, $
    Image = image, $
    BackColor=backcolor, $
    AnnotateColor=annotatecolor, $
    ColorTable=colortable, $
    NColors=ncolors, $
    Position=position, $
    Vertical=vertical, $
    XScale=xscale, $
    XTitle=xtitle, $
    YScale=yscale, $
    YTitle=ytitle
; This method gets the properties of the object.
; Error handling.
Catch, theError
IF theError NE 0 THEN BEGIN
    Catch, /Cancel
    ok = Error_Message(!Error_State.Msg + ' Returning...', $
        Traceback=1, /Error)
    RETURN
ENDIF
; Set properties if keyword is present.
IF Arg_Present(colortable) THEN colortable = self.colortable
IF Arg_Present(backcolor) THEN backcolor = self.backcolor
IF Arg_Present(annotatecolor) THEN annotatecolor = self.annotatecolor
IF Arg_Present(ncolors) THEN ncolors = self.ncolors
IF Arg_Present(vertical) THEN vertical = self.vertical
IF Arg_Present(xtitle) THEN xtitle = self.xtitle
IF Arg_Present(ytitle) THEN ytitle = self.ytitle
IF Arg_Present(xscale) THEN xscale = self.xscale
IF Arg_Present(yscale) THEN yscale = self.yscale
IF Arg_Present(image) THEN image = *self.image
END
```

```
FUNCTION BoxImage::GetAnnotationColor
; This method returns the annotation color.
```

Appendix C: IDL Program Code

```
RETURN, self.annotatecolor
END


---



---



```
PRO BoxImage::SetProperty, $
 Image = image, $
 AnnotateColor=annotatecolor, $
 BackColor=backcolor, $
 ColorTable=colortable, $
 Draw=draw, $
 NColors=ncolors, $
 Position=position, $
 Vertical=vertical, $
 XScale=xscale, $
 XTitle=xtitle, $
 YScale=yscale, $
 YTitle=ytitle, $
 _Extra=extra
; This method sets the properties of the object.
; Error handling.
Catch, theError
IF theError NE 0 THEN BEGIN
 Catch, /Cancel
 ok = Error_Message(!Error_State.Msg + ' Returning...', $
 Traceback=1, /Error)
 RETURN
ENDIF
; Set properties if keyword is present.
IF N_Elements(backcolor) NE 0 THEN self.backcolor = backcolor
IF N_Elements(annotatecolor) NE 0 THEN self.annotatecolor = annotatecolor
IF N_Elements(ncolors) NE 0 THEN self.ncolors = ncolors
IF N_Elements(position) NE 0 THEN self.position = position
IF N_Elements(vertical) NE 0 THEN self.vertical = vertical
IF N_Elements(xtitle) NE 0 THEN self.xtitle = xtitle
IF N_Elements(ytitle) NE 0 THEN self.ytitle = ytitle
IF N_Elements(xscale) NE 0 THEN self.xscale = xscale
IF N_Elements(yscale) NE 0 THEN self.yscale = yscale
IF N_Elements(extra) NE 0 THEN *self.extra = extra
IF N_Elements(image) NE 0 THEN BEGIN
 *self.image = image
 *self.process = image
 *self.undo = image
ENDIF
IF N_Elements(colortable) NE 0 THEN BEGIN
 colors = Obj_New("IDLgrPalette")
 colors->LoadCT, 0 > colortable < 40
 colors->GetProperty, Red=r, Green=g, Blue=b
 Obj_Destroy, colors
 *self.r = Congrid(r, self.ncolors)
 *self.g = Congrid(g, self.ncolors)
 *self.b = Congrid(b, self.ncolors)
ENDIF
IF Keyword_Set(draw) THEN self->Draw
END

```


```

```
PRO BoxImage::LoadCT, colortable, Draw=draw
```

```

; This method loads a different color table for the image.

IF N_Elements(colortable) EQ 0 THEN BEGIN
  colors = Obj_New("IDLgrPalette")
  colors->LoadCT, 0
  colors->GetProperty, Red=r, Green=g, Blue=b
  Obj_Destroy, colors
ENDIF ELSE BEGIN
  colors = Obj_New("IDLgrPalette")
  colors->LoadCT, 0 > colortable < 40
  colors->GetProperty, Red=r, Green=g, Blue=b
  Obj_Destroy, colors
ENDELSE
*self.r = Congrid(r, self.ncolors)
*self.g = Congrid(g, self.ncolors)
*self.b = Congrid(b, self.ncolors)

; Redraw the image if needed.

IF Keyword_Set(draw) THEN self->Draw
END

```

```

PRO BoxImage:::Draw, Font=font
; This method draws the graphics display.

; Error handling.

Catch, theError
IF theError NE 0 THEN BEGIN
  Catch, /Cancel
  ok = Error_Message(!Error_State.Msg + ' Returning...', $
    Traceback=1, /Error)
  RETURN
ENDIF

; Check keywords.

IF N_Elements(font) EQ 0 THEN font = !P.Font
; Obtain the annotate color.

annotateColor = GetColor(self.annotatecolor, !D.Table_Size-2)
backColor = GetColor(self.backColor, !D.Table_Size-3)
; Load the image colors.

TVLCT, *self.r, *self.g, *self.b
; Calculate the position of the image and color bar
; in the window.

IF self.vertical THEN BEGIN
  p = self.position
  length = p[2] - p[0]
  imgpos = [p[0], p[1], (p[2]-(0.20*length)), p[3]]
  cbpos = [(p[0]+(0.93*length)), p[1], p[2], p[3]]
ENDIF ELSE BEGIN
  p = self.position
  length = p[3] - p[1]
  imgpos = [p[0], p[1], p[2], (p[3]-(0.20*length))]
  cbpos = [p[0], (p[1]+(0.93*length)), p[2], p[3]]
ENDELSE

; Need to erase display? Only on displays with windows.

IF (!D.Flags AND 256) NE 0 THEN Erase, Color=backColor
; Calculate appropriate character size for plots.

thisCharsize = Str_Size('A Sample String', 0.25)

```

Appendix C: IDL Program Code

```

; Draw the graphics.

TVImage, BytScl(*self.process, Top=self.ncolors-1), $
Position=imgpos, _Extra=*self.extra
Plot, self.xscale, self.yscale, XStyle=1, YStyle=1, $
XTitle=self.xtitle, YTitle=self.ytitle, Color=annotateColor, $
Position=imgpos, /NoErase, /NoData, Ticklen=-0.025, _Extra=*self.extra, $
CharSize=thisCharSize, Font=font
Colorbar, Range=[Min(*self.process), Max(*self.process)], Divisions=8, $
_Extra=*self.extra, Color=annotateColor, Position=cbpos, Ticklen=-0.2, $
Vertical=self.vertical, NColors=self.ncolors, CharSize=thisCharSize, Font=font
END

```

```

Function BoxImage::Init, $           ; The name of the method.
    image, $                      ; The image data.
    AnnotateColor=annotatecolor, $ ; The annotation color.
    BackColor=backcolor, $        ; The background color.
    ColorTable=colortable, $      ; The colortable index.
    NColors=ncolors, $            ; Number of image colors.
    Position=position, $          ; Position in window.
    Vertical=vertical, $          ; Vertical colorbar flag.
    XScale=xscale, $              ; The scale on X axis.
    XTitle=xtitle, $              ; The title on X axis.
    YScale=yscale, $              ; The scale on Y axis.
    YTitle=ytitle, $              ; The title on Y axis.
    _Extra=extra                  ; Holds extra keywords.

; The initialization routine for the object. Create the
; particular instance of the object class.

; Error handling.

Catch, theError
IF theError NE 0 THEN BEGIN
    Catch, /Cancel
    ok = Error_Message(!Error_State.Msg + ' Returning...', $
        Traceback=1, /Error)
    RETURN, 0
ENDIF

; Check for positional parameter. Define if necessary.

IF N_Elements(image) EQ 0 THEN image = LoadData(7)
ndims = Size(image, /N_Dimensions)
IF ndims NE 2 THEN Message, 'Image must be 2D array.', /NoName

; Check for keyword parameters.

IF N_Elements(annotatecolor) EQ 0 THEN annotatecolor = "NAVY"
IF N_Elements(backcolor) EQ 0 THEN backcolor = "WHITE"
IF N_Elements(ncolors) EQ 0 THEN ncolors = !D.Table_Size - 3
IF N_Elements(position) EQ 0 THEN position = [0.15, 0.15, 0.9, 0.9]
vertical = Keyword_Set(vertical)
IF N_Elements(xtitle) EQ 0 THEN xtitle = ""
IF N_Elements(ytitle) EQ 0 THEN ytitle = ""

s = Size(image, /Dimensions)
IF N_Elements(xscale) EQ 0 THEN xscale = [0,s[0]]
IF N_Elements(yscale) EQ 0 THEN yscale = [0,s[1]]

IF N_Elements(colortable) EQ 0 THEN BEGIN
    colors = Obj_New("IDLgrPalette")
    colors->LoadCT, 0
    colors->GetProperty, Red=r, Green=g, Blue=b
    Obj_Destroy, colors
ENDIF ELSE BEGIN
    colors = Obj_New("IDLgrPalette")
    colors->LoadCT, 0 > colortable < 40

```

```

colors->GetProperty, Red=r, Green=g, Blue=b
Obj_Destroy, colors
ENDELSE

r = Congrid(r, ncolors)
g = Congrid(g, ncolors)
b = Congrid(b, ncolors)

; Populate the object.

self.image = Ptr_New(image)
self.process = Ptr_New(image)
self.undo = Ptr_New(image)
self.position = position
self.ncolors = ncolors
self.annotatecolor = annotatecolor
self.backcolor = backcolor
self.r = Ptr_New(r)
self.g = Ptr_New(g)
self.b = Ptr_New(b)
self.vertical = vertical
self.xscale = xscale
self.yscale = yscale
self.xtitle = xtitle
self.ytitle = ytitle
self.extra = Ptr_New(extra)

; Indicate successful initialization.

RETURN, 1
END

```

```

PRO BoxImage::Cleanup
; The clean-up routine for the object. Free all
; pointers.

Ptr_Free, self.image
Ptr_Free, self.process
Ptr_Free, self.undo
Ptr_Free, self.r
Ptr_Free, self.g
Ptr_Free, self.b
Ptr_Free, self.extra
END

```

```

PRO BoxImage__Define
; The definition of the BOXIMAGE object class.

struct = { BOXIMAGE, $           ; The BOXIMAGE object class.
          image: Ptr_New(), $    ; The original image data.
          process: Ptr_New(), $   ; The processed image data.
          undo: Ptr_New(), $     ; The previous processed image data.
          position: FltArr(4), $  ; The position of the graphics output in window.
          r: Ptr_New(), $        ; The red color vector associated with image colors.
          g: Ptr_New(), $        ; The green color vector associated with image colors.
          b: Ptr_New(), $        ; The blue color vector associated with image colors.
          ncolors: 0L, $         ; The number of image colors.
          annotatecolor: "", $   ; The name of the annotation color.
          backcolor: "", $       ; The name of the background color.
          xscale: FltArr(2), $    ; The scale for the X axis of the image plot.
          yscale: FltArr(2), $    ; The scale for the Y axis of the image plot.

```

```

        xtitle: "", $           ; The title of the X axis.
        ytitle: "", $           ; The title of the Y axis.
        vertical: 0L, $         ; A flag to indicate a vertical color bar.
        extra: Ptr_New() $      ; A placeholder for extra keywords.
    }
END

```

HDFRead Program

This is an example program for reading an HDF file that is created with *HDFWrite*. It demonstrates how to find and access both the data and the data attributes in an HDF file. Additional information about HDF files can be found in “Reading and Writing HDF Data” on page 161. The file can be downloaded from the *coyote* anonymous ftp site. The URL is:

<ftp://ftp.dfanning.com/pub/dfanning/outgoing/coyote2nd/hdfread.pro>

```

PRO HDFREAD, filename
    ; Open file and initialize the SDS interface.
    IF N_ELEMENTS(filename) EQ 0 THEN filename = DIALOG_PICKFILE(File='example.hdf')
    IF NOT HDF_ISHDF(filename) THEN BEGIN
        PRINT, 'Invalid HDF file ...'
        RETURN
    ENDIF ELSE PRINT, 'Valid HDF file. Opening "' + filename + '"'
    newFileID = HDF_SD_START(filename, /READ)

    ; What is in the file. Print the number of
    ; datasets, attributes, and palettes.

    PRINT, 'Reading number of datasets/attributes in file ...'
    HDF_SD_FILEINFO, newFileID, datasets, attributes
    numPalettes = HDF_DFP_NPALS(filename)
    PRINT, ''
    PRINT, 'No. of Datasets: ', datasets
    PRINT, 'No. of Attributes: ', attributes
    PRINT, 'No. of Palettes: ', numPalettes
    ; Print the name of each SDS.
    PRINT, ''
    FOR j=0, datasets-1 DO BEGIN
        thisSDS = HDF_SD_SELECT(newFileID, j)
        HDF_SD_GETINFO, thisSDS, NAME=thisSDSName
        PRINT, 'Dataset No. ', STRTRIM(j, 2), ': ', thisSDSName
    ENDFOR
    ; Print the name of each attribute.

    PRINT, ''
    FOR j=0, attributes-1 DO BEGIN
        HDF_SD_ATTRINFO, newFileID, j, NAME=thisAttr
        PRINT, 'File Attribute No. ', + STRTRIM(j, 2), ': ', thisAttr
    ENDFOR
    ; Find the index of the "Gridded Data" SDS.

    index = HDF_SD_NAMETOINDEX(newFileID, "Gridded Data")
    ; Select the Gridded Data SDS.

    thisSdsID = HDF_SD_SELECT(newFileID, index)
    ; Print the names of the Gridded Data attributes.

    PRINT, ''
    HDF_SD_GETINFO, thisSdsID, NATTS=numAttributes
    PRINT, 'Number of Gridded Data attributes: ', numAttributes
    FOR j=0,numAttributes-1 DO BEGIN

```

```

HDF_SD_ATTRINFO, thisSdsID, j, NAME=thisAttr
PRINT, 'SDS Attribute No. ',+STRTRIM(j, 2), ': ', thisAttr
ENDFOR

; Get the data.

PRINT, ''
PRINT, 'Reading gridded data ...'
HDF_SD_GETDATA, thisSdsID, newGriddedData
; Get the palette associated with this data.

PRINT, 'Reading the color palette ...'
HDF_DFP_GETPAL, filename, thisPalette
; Get the gridded DIMENSION data.

longitudeDimID = HDF_SD_DIMGETID(thisSdsID, 0)
latitudeDimID = HDF_SD_DIMGETID(thisSdsID, 1)
PRINT, 'Reading the dimension data ...'
HDF_SD_DIMGET, longitudeDimID, LABEL=lonlable, SCALE=lonscale, UNIT=lonunits
HDF_SD_DIMGET, latitudeDimID, LABEL=latlable, SCALE=latscale, UNIT=latunits

; Get the DATE and EXPERIMENT attributes.

PRINT, 'Reading file attributes for plot ...'
dateAttID = HDF_SD_ATTRFIND(newFileID, 'DATE')
expAttID = HDF_SD_ATTRFIND(newFileID, 'EXPERIMENT')
HDF_SD_ATTRINFO, newFileID, dateAttID, DATA=thisDate
HDF_SD_ATTRINFO, newFileID, expAttID, DATA=thisExperiment

; Draw a contour map of the data.

PRINT, 'Drawing contour plot ...'
WINDOW, XSIZE=400, YSIZE=400
TVLCT, TRANSPOSE(CONGRID(thisPalette, 3, !D.Table_Size-1))
TVIMAGE, BYTSCl(newGriddedData, TOP=!D.Table_Size-1), $
    POSITION=[0.15, 0.15, 0.95, 0.87]
CONTOUR, newGriddedData, lonscale, latscale, $
    XSTYLE=1, YSTYLE=1, NLEVELS=14, /NOERASE, $
    XTITLE = lonlable + ' (' + lonunits + ')', $
    YTITLE = latlable + ' (' + latunits + ')', $
    TITLE = thisExperiment + ' on ' + thisDate, $
    POSITION=[0.15, 0.15, 0.95, 0.87], /FOLLOW, $
    CHARSIZE=1.25, C_COLOR=0
HDF_SD_END, newFileID

PRINT, 'Read operation complete.'
END

```

HDFWrite Program

This is an example program for writing an HDF file that can be read with *HDFRead*. It demonstrates how to create SDS data sets and both data and file attributes in an HDF file. Additional information about HDF files can be found in “Reading and Writing HDF Data” on page 161. The file can be downloaded from the *coyote* anonymous ftp site. The URL is:

<ftp://ftp.dfanning.com/pub/dfanning/outgoing/coyote2nd/hdfwrite.pro>

```

PRO HDFWRITE, filename
; Create randomly-distributed data.

seed = -1L
x = RANDOMU(seed, 40)
y = RANDOMU(seed, 40)
distribution = SHIFT(DIST(40,40), 25, 15)
distribution = EXP(-(distribution/15)^2)

```

Appendix C: IDL Program Code

```

lat = x * (24./1.0) + 24
lon = y * 50.0/1.0 - 122
temp = distribution(x*40, y*40) * 273
      ; Select the name of a new file and open it.

IF N_ELEMENTS(filename) EQ 0 THEN $
  filename = DIALOG_PICKFILE(/Write, File='example.hdf')
IF filename EQ '' THEN RETURN
PRINT, ''
PRINT, 'Opening HDF file "' + filename + '" ...'
      ; If there is a problem opening the file, catch the error.

CATCH, error
IF error NE 0 THEN BEGIN
  PRINT, ''
  PRINT, 'Unable to obtain an SDS file ID.'
  PRINT, 'This file may already be open by an HDF routine.'
  PRINT, 'Returning...'
  PRINT, ''
  RETURN
ENDIF
fileID = HDF_SD_START(filename, /CREATE)
CATCH, /CANCEL

      ; Write some attributes into the file.

PRINT, 'Writing file attributes ...'
version = 'MacOS 4.0.1b'
date = "Jan 1, 1997"
experiment = "Experiment 25A36M"
name = 'David Fanning'
email = 'davidf@dfanning.com'
HDF_SD_ATTRSET, fileID, 'VERSION', version
HDF_SD_ATTRSET, fileID, 'DATE', date
HDF_SD_ATTRSET, fileID, 'EXPERIMENT', experiment
HDF_SD_ATTRSET, fileID, 'NAME', name
HDF_SD_ATTRSET, fileID, 'EMAIL ADDRESS', email

      ; Create the SDS data sets for the raw data.

PRINT, 'Writing raw data ...'
latsdsID = HDF_SD_CREATE(fileID, "Raw Latitude", [40L], /FLOAT)
lonsdsID = HDF_SD_CREATE(fileID, "Raw Longitude", [40L], /FLOAT)
tempsdsID = HDF_SD_CREATE(fileID, "Raw Temperature", [40L], /FLOAT)

      ; Write the raw data.

HDF_SD_ADDDATA, latsdsID, lat
HDF_SD_ADDDATA, lonsdsID, lon
HDF_SD_ADDDATA, tempsdsID, temp

      ; Terminate access to the raw data SDSS.

HDF_SD_ENDACCESS, latsdsID
HDF_SD_ENDACCESS, lonsdsID
HDF_SD_ENDACCESS, tempsdsID

      ; Grid the irregularly spaced, raw data.

latMax = 49.0
latMin = 24.0
lonMax = -67.0
lonMin = -125.0
mapBounds = [lonMin, latMin, lonMax, latMax]
mapSpacing = [0.5, 0.25]
TRIANGULATE, lon, lat, FVALUE=temp, SPHERE=triangles, /DEGREES
gridData = TRIGRID(temp, SPHERE=triangles, /DEGREES, $
  /EXTRAPOLATE, mapSpacing, mapBounds)

      ; Calculate vectors corresponding to gridded data.

s = SIZE(gridData)

```

```

gridlon = FINDGEN(s(1))*((lonMax - lonMin)/(s(1)-1)) + lonMin
gridlat = FINDGEN(s(2))*((latMax - latMin)/(s(2)-1)) + latMin
; Now store the gridded data.

PRINT, 'Writing gridded data ...'
gridID = HDF_SD_CREATE(fileID, "Gridded Data", [s(1), s(2)], /FLOAT)
HDF_SD_ADDDATA, gridID, gridData
; Store local SDS attributes with the gridded data.

method = 'Delauney Triangulation'
routines = 'TRIANGULATE, TRIGRID'
PRINT, 'Writing gridded data set attributes ...'
HDF_SD_AttrSet, gridID, 'METHOD', method
HDF_SD_AttrSet, gridID, 'IDL ROUTINES', routines
HDF_SD_AttrSet, gridID, 'GRID SPACING', mapSpacing
HDF_SD_AttrSet, gridID, 'MAP BOUNDARIES', mapBounds
; Store the scales for the dimensions of the gridded data.

PRINT, 'Writing dimension data ...'
longitudeDimID = HDF_SD_DIMGETID(gridID, 0)
HDF_SD_DIMSET, longitudeDimID, $
LABEL='Longitude', $
NAME='Longitude Dimension', $
SCALE=gridlon, $
UNIT='Degrees'
latitudeDimID = HDF_SD_DIMGETID(gridID, 1)
HDF_SD_DIMSET, latitudeDimID, $
LABEL='Latitude', $
NAME='Latitude Dimension', $
SCALE=gridlat, $
UNIT='Degrees'
; Load a color palette you will use to display the data.

thisDevice = !D.NAME
SET_PLOT, 'Z'
LOADCT, 5, /SILENT
TVLCT, r, g, b, /GET
palette = BYTARR(3,256)
palette(0,*) = r
palette(1,*) = g
palette(2,*) = b
SET_PLOT, thisDevice
; Store the color palette along with the data.

PRINT, 'Writing color palette ...'
HDF_DFP_ADDPAL, filename, palette
; Close everything up properly.

HDF_SD_ENDACCESS, gridID
HDF_SD_END, fileID
PRINT, 'Write Operation Completed'
PRINT, ''
END

```

HistoImage Program

This is an example of a graphics display program. It is described in detail in the section “The HistoImage Program” on page 240. The source code for this program can be downloaded from the *Coyote’s Guide to IDL Programming* anonymous ftp site. The URL is:

<ftp://ftp.dfanning.com/pub/dfanning/outgoing/coyote2nd/histoimage.pro>

```

PRO HistoImage, $
  image, $                                ; The program name.
  AxisColorName=axisColorName, $             ; The image data.
  BackColorName=backcolorName, $              ; The axis color.
  Binsize=binsize, $                         ; The background color.
  ColorTable=colortable, $                   ; The histogram bin size.
  DataColorName=datacolorName, $              ; The colortable index to load.
  Debug=debug, $                            ; The data color.
  Extra=extra, $                           ; A debug flag variable.
  _ImageColors=imagecolors, $                ; For passing extra keywords.
  Max_Value=max_value, $                   ; The number of image colors used. (Out)
  NoLoadCT=noloadct, $                     ; The maximum value of the histogram plot.
  XScale=xscale, $                         ; A flag to not load the image color table.
  YScale=yscale                            ; The scale for the X axis of the image.
  ; The scale for the Y axis of the image.

; Catch any error in the HistoImage program.

Catch, theError
IF theError NE 0 THEN BEGIN
  Catch, /Cancel
  ok = Error_Message(!Error_State.Msg + ' Returning...', $
    Traceback=Keyword_Set(debug))
  RETURN
ENDIF

; Check for positional parameter. Define if necessary.
; Make sure it is correct size.

IF N_Elements(image) EQ 0 THEN image = LoadData(7)
ndim = Size(image, /N_Dimensions)
IF ndim NE 2 THEN Message, '2D Image Variable Required.', /NoName
; Check for histogram keywords.

IF N_Elements(binsize) EQ 0 THEN BEGIN
  range = Max(image) - Min(image)
  binsize = 2.0 > (range / 128.0)
ENDIF

IF N_Elements(max_value) EQ 0 THEN max_value = 5000.0
; Check for image scale parameters.

s = Size(image, /Dimensions)
IF N_Elements(xscale) EQ 0 THEN xscale = [0, s[0]]
IF N_Elements(xscale) NE 2 THEN Message, 'XSCALE must be 2-element array', /NoName
IF N_Elements(yscale) EQ 0 THEN yscale = [0, s[1]]
IF N_Elements(yscale) NE 2 THEN Message, 'YSCALE must be 2-element array', /NoName
; Check for color keywords.

IF N_Elements(dataColorName) EQ 0 THEN dataColorName = "Red"
IF N_Elements(axisColorName) EQ 0 THEN axisColorName = "Navy"
IF N_Elements(backColorName) EQ 0 THEN backColorName = "White"
IF N_Elements(colortable) EQ 0 THEN colortable = 4
colortable = 0 > colortable < 40
imagecolors = !D.Table_Size-4
; Load plot colors.

axisColor = GetColor(axisColorName, !D.Table_Size-2)
dataColor = GetColor(datacolorName, !D.Table_Size-3)
backColor = GetColor(backColorName, !D.Table_Size-4)

; I don't always want to load a color table. Sometimes I
; want to control colors outside the program. Check
; the NOLoadCT keyword before loading.

IF NOT Keyword_Set(noloadct) THEN BEGIN
  LoadCT, colortable, NColors=imagecolors, /Silent
ENDIF ELSE BEGIN
; This code placed here to work around an obscure

```

```

; PRINTER bug in IDL 5.3 that causes all pixels with
; value 0 to be displayed in the last single color
; loaded (the backColor, in this case).
IF !D.NAME EQ 'PRINTER' THEN BEGIN
    ; Just get the color table vectors and re-load the colors.
    TVLCT, r, g, b, /Get
    TVLCT, r, g, b
ENDIF
ENDELSLE
; Determine positions of graphics in window.

histoPos = [0.15, 0.675, 0.95, 0.950]
colorbarPos = [0.15, 0.500, 0.95, 0.550]
imagePos = [0.15, 0.100, 0.95, 0.400]

; Calculate appropriate character size for plots.

thisCharSize = Str_Size('A Sample String', 0.20)

; Calculate the histogram.

histdata = Histogram(image, Binsize=binsize, Max=Max(image), Min=Min(image))

; Have to fudge the bins and histdata variables to get the
; histogram plot to make sense.

npts = N_Elements(histdata)
halfbinsize = binsize / 2.0
bins = Findgen(N_Elements(histdata)) * binsize + Min(image)
binsToPlot = [bins[0], bins + halfbinsize, bins[npts-1] + binsize]
histdataToPlot = [histdata[0], histdata, histdata[npts-1]]
xrange = [Min(binsToPlot), Max(binsToPlot)]

; Plot the histogram of the display image. Axes first.

Plot, binsToPlot, histdataToPlot, $ ; The fudged histogram and bin data.
Background=backcolor, $ ; The background color of the display.
CharSize=thisCharSize, $ ; The character size.
Color=axiscolor, $ ; The color of the axes.
Max_Value=max_value, $ ; The maximum value of the plot.
NoData=1, $ ; Draw the axes only. No data.
Position=histoPos, $ ; The position of the plot in the window.
Title='Image Histogram', $ ; The title of the plot.
XRange=xrange, $ ; The X data range.
XStyle=1, $ ; Exact axis scaling. No autoscaled axes.
XTickFormat='(I6)', $ ; The format of the X axis annotations.
XTitle='Image Value', $ ; The title of the X axis.
YMinor=1, $ ; One minor tick mark on X axis.
YRange=[0,max_value], $ ; The Y data range.
YStyle=1, $ ; Exact axis scaling. No autoscaled axes.
YTickFormat='(I6)', $ ; The format of the Y axis annotations.
YTitle='Pixel Density', $ ; The title of the Y axis.
_Extra=extra ; Pass any extra PLOT keywords.

; Overplot the histogram data in the data color.

OPlot, binsToPlot, histdataToPlot, PSym=10, Color=dataColor
; Make histogram boxes by drawing lines in data color.

FOR j=1L,N_Elements(bins)-2 DO BEGIN
    PlotS, [bins[j], bins[j]], [!Y.CRange[0], histdata[j] < max_value], $
        Color=dataColor
ENDFOR
; Display the colorbar.

cbarRange = [Min(binsToPlot), Max(binsToPlot)]
Colorbar, $
    CharSize=thisCharSize, $ ; The character size as determined by Str_Size.
    Color=axisColor, $ ; The annotation is the axis color.
    Divisions=0, $ ; Use default PLOT divisions by setting to 0.

```

```

NColors=imagecolors, $      ; The number of image colors.
Position=colorbarPos, $    ; The position of the colorbar in the window.
Range=cbarRange, $          ; The range of the color bar.
XTicklen=-0.2, $            ; Outward facing tick marks.
_Extra=extra                ; Any extra COLORBAR keywords.

; Display the image.

TVImage, BytScl(image, Top=imagecolors-1), Position=imagePos, _Extra=extra

Plot, xscale,yscale, $
  CharSize=thisCharSize, $ ; The character size as determined by Str_Size.
  Color=axisColor, $       ; The outline should be in the axes color.
  NoData=1, $              ; No data. Draw axes only.
  NoErase=1, $              ; Don't erase what is already in the window.
  Position=imagePos, $     ; The position of the axes around the image.
  XStyle=1, $               ; No axis autoscaling.
  XTicklen=-0.025, $        ; Outward facing tick marks.
  YStyle=1, $               ; No axis autoscaling
  YTicklen=-0.025, $        ; Outward facing tick marks.
  _Extra=extra                ; Any extra PLOT keywords.

END

```

Histo_GUI Program

This is an example of a widget program. It is the final *Histo_GUI* program that is described in the chapters *Writing a Widget Program*, *Widget Programming Techniques*, and the first part of *Creating Dialog Form Widgets*. The source code for this program can be downloaded from the *coyote* anonymous ftp site. The URL is:

ftp://ftp.dfanning.com/pub/dfanning/outgoing/coyote2nd/histo_gui.7.pro

```

PRO Histo_GUI_Open_Image, event
; This event handler opens and displays a new image.

; Bad things can happen here!! Good error handling
; is essential.

Catch, theError
IF theError NE 0 THEN BEGIN
  Catch, /Cancel
  ok = Error_Message(Traceback=1)
  IF N_Elements(info) NE 0 THEN $
    Widget_Control, event.top, Set_UValue=info, /No_Copy
  RETURN
ENDIF

; Gather information from the user about the image file.

Widget_Control, event.top, KBRD_Focus_Events=0
 fileInfo = OpenImage(Cancel=cancelled, Group_Leader=event.top)
Widget_Control, event.top, KBRD_Focus_Events=1
IF cancelled THEN RETURN

; Alright. Read the image data.

newimage = BytArr(fileInfo.xsize, fileInfo.ysize)
OpenR, lun, fileInfo.filename, /Get_Lun
ReadU, lun, newimage
Free_Lun, lun

; Get the info structure.

Widget_Control, event.top, Get_UValue=info, /No_Copy
; Store the new image in the info structure.

*info.image = newimage

```

```

*info.process = newimage
*info.undo = newimage
; No way to UNDO. Turn undo button off.
Widget_Control, info.undoID, Sensitive=0
; Redisplay the image data.

WSet, info.pixID
HistoImage, *info.process, $
AxisColorName=info.axisColorName, $
BackColorName=info.backcolorName, $
Binsize=info.binsize, $
DataColorName=info.datacolorName, $
_Extra=*info.extra, $
_Max_Value=info.max_value, $
NoLoadCT=1, $
XScale=info.xscale, $
YScale=info.yscale
WSet, info.wid
Device, Copy=[0, 0, !D.X_Size, !D.Y_Size, 0, 0, info.pixID]
Widget_Control, event.top, Set_UValue=info, /No_Copy
END
=====
```

```

PRO Histo_GUI_Print, event
; This event handler sends output to the default printer.
; Which printer? How many copies? Etc.
ok = Dialog_PrinterSetup()
IF NOT ok THEN RETURN
; Get the info structure.
Widget_Control, event.top, Get_UValue=info, /No_Copy
; Set up for printing.
thisDevice = !D.Name
Device, Get_Visual_Depth=theDepth
thisFont = !P.Font
thickness = !P.Thick
!P.Font=1
!P.Thick = 2
Widget_Control, /Hourglass
; Portrait or Landscape mode?
Widget_Control, event.id, Get_Value=buttonValue
CASE buttonValue OF
'Portrait Mode': BEGIN
    keywords = PSWindow(/Printer, Fudge=0.25)
    Set_Plot, 'PRINTER', /Copy
    Device, Portrait=1
    ENDCASE
'Landscape Mode': BEGIN
    keywords = PSWindow(/Printer, /Landscape, Fudge=0.25)
    Set_Plot, 'PRINTER', /Copy
    Device, Landscape=1
    ENDCASE
ENDCASE
; Configure the Printer device.
Device, _Extra=keywords
; Stretch the color table vectors if on 8-bit display.
```

Appendix C: IDL Program Code

```
IF theDepth EQ 8 THEN BEGIN
    topColor = info.imagecolors-1
    TVLCT, Congrid(info.r[0:topColor], !D.Table_Size-4), $
        Congrid(info.g[0:topColor], !D.Table_Size-4), $
        Congrid(info.b[0:topColor], !D.Table_Size-4)
ENDIF
; Draw the graphics display. No drawing color keywords will
; put default drawing colors into effect.

HistoImage, *info.process, $
AxisColorName='Black', $
Binsize=info.binsize, $
DataColorName='Black', $
_Extra=*info.extra, $
Max_Value=info.max_value, $
NoLoadCT=1, $
XScale=info.xscale, $
YScale=info.yscale
; Close the printer. Clean up.

Device, /Close_Document
Set_Plot, thisDevice
!P.Font = thisFont
!P.Thick = thickness
Widget_Control, event.top, Set_UValue=info, /No_Copy
END
```

```
PRO Histo_GUI_File_Output, event
; This event handler creates output files of various sorts.
; Get visual depth and decomposition state.

Device, Get_Visual_Depth=theDepth, Get_Decomposed=theState
; Focus events off for 24-bit displays.

IF theDepth GT 8 THEN Widget_Control, event.top, KBRD_Focus_Events=0
; Gather button information. Construct default filename.

Widget_Control, event.id, Get_Value=buttonValue, Get_UValue=file_extension
startFilename = 'histo_gui' + file_extension
; Get either the output filename or the PostScript device keywords
; before you get the info structure.

IF buttonValue EQ 'PostScript File' THEN BEGIN
    keywords = PSConfig(Cancel=cancelled, Filename=startFilename, Group_Leader=event.top)
    IF cancelled THEN RETURN
ENDIF ELSE BEGIN
    filename = Dialog_Pickfile(File=startFilename, /Write)
    IF filename EQ "" THEN RETURN
ENDIFELSE
; Turn keyboard focus events back on for 24-bit displays.

IF theDepth GT 8 THEN Widget_Control, event.top, KBRD_Focus_Events=1
; Get the info structure.

Widget_Control, event.top, Get_UValue=info, /No_Copy
; Make sure we know which window we are copying information from.

WSet, info.wid
; What kind of file do you want to make? Must do different things
; on different depth displays.
```

```

CASE buttonValue OF
  'GIF File': BEGIN
    IF theDepth GT 8 THEN BEGIN
      Device, Decomposed=1
      snapshot = TVRD(True=1)
      Device, Decomposed=theState
      image2D = Color_Quan(snapshot, 1, r, g, b, Colors=256, /Dither)
    ENDIF ELSE BEGIN
      TVLCT, r, g, b, /Get
      image2D = TVRD()
    ENDELSE
    Write_GIF, filename, image2D, r, g, b
  END

  'JPEG File': BEGIN
    IF theDepth GT 8 THEN BEGIN
      Device, Decomposed=1
      image3D = TVRD(True=1)
      Device, Decomposed=theState
    ENDIF ELSE BEGIN
      image2D = TVRD()
      TVLCT, r, g, b, /Get
      s = Size(image2D, /Dimensions)
      image3D = BytArr(3, s[0], s[1])
      image3D[0,*,*] = r[image2d]
      image3D[1,*,*] = g[image2d]
      image3D[2,*,*] = b[image2d]
    ENDELSE
    Write_JPEG, filename, image3D, True=1, Quality=85
  END

  'TIFF File': BEGIN
    IF theDepth GT 8 THEN BEGIN
      Device, Decomposed=1
      image3D = TVRD(True=1)
      Device, Decomposed=theState
    ENDIF ELSE BEGIN
      image2D = TVRD()
      TVLCT, r, g, b, /Get
      s = Size(image2D, /Dimensions)
      image3D = BytArr(3, s[0], s[1])
      image3D[0,*,*] = r[image2d]
      image3D[1,*,*] = g[image2d]
      image3D[2,*,*] = b[image2d]
    ENDELSE
    Write_TIFF, filename, Reverse(Temporary(image3D), 3)
  END

  'PostScript File': BEGIN
    ; Store the device name and the current font.
    thisDevice = !D.Name
    thisFont = !P.Font
    ; Use hardware fonts for the PostScript file.
    !P.Font = 0
    Set_Plot, 'PS'
    ; Have to resample colors if running on 8-bit display.
    IF theDepth EQ 8 THEN BEGIN
      topColor = info.imagecolors-1
      TVLCT, Congrid(info.r[0:topColor], !D.Table_Size-4), $
        Congrid(info.g[0:topColor], !D.Table_Size-4), $
        Congrid(info.b[0:topColor], !D.Table_Size-4)
    ENDIF
  END

```

Appendix C: IDL Program Code

```
; Configure the PostScript device.  
Device, _Extra=keywords  
; Draw the graphics display. No drawing color keywords will  
; put default drawing colors into effect.  
HistoImage, *info.process, $  
Binsize=info.binsize, $  
_Extra=*info.extra, $  
Max_Value=info.max_value, $  
NoLoadCT=1, $  
XScale=info.xscale, $  
YScale=info.yscale  
; Close the PostScript file and clean up.  
Device, /Close_File  
Set_Plot, thisDevice  
!P.Font = thisFont  
END  
ENDCASE  
Widget_Control, event.top, Set_UValue=info, /No_Copy  
END
```

```
PRO Histo_GUI_Image_Colors, event  
; This event handler changes the image colors of the graphic display.  
Widget_Control, event.top, Get_UValue=info, /No_Copy  
; What kind of event is this: button or color table loading?  
thisEvent = Tag_Names(event, /Structure_Name)  
CASE thisEvent OF  
'WIDGET_BUTTON': BEGIN  
; Load the current color vectors.  
TVLCT, info.r, info.g, info.b  
; Create an unique title for the XColors program. Assign  
; it to the top-level base widget.  
colorTitle = info.title + " (" + StrTrim(info.wid,2) + ")"  
Widget_Control, event.top, TLB_Set_Title = colorTitle  
; Call XColors with NOTIFYID to alert widgets when colors change.  
XColors, NColors=info.imagecolors, Group_Leader=event.top, $  
NotifyID=[event.id, event.top], Title=colortitle + ' Colors'  
END  
'XCOLORS_LOAD': BEGIN  
; Update the color vectors with new values.  
info.r = event.r  
info.g = event.g  
info.b = event.b  
; Redisplay the graphic if necessary.  
Device, Get_Visual_Depth=theDepth  
IF theDepth GT 8 THEN BEGIN  
; Make the pixmap window the current graphics window.  
WSet, info.pixID
```

```

; Draw the graphics.

HistoImage, *info.process, $
AxisColorName=info.axisColorName, $
BackColorName=info.backcolorName, $
Binsize=info.binsize, $
DataColorName=info.datacolorName, $
_Extra=*info.extra, $
Max_Value=info.max_value, $
NoLoadCT=1, $
XScale=info.xscale, $
YScale=info.yscale
WSet, info.wid
Device, Copy=[0, 0, !D.X_Size, !D.Y_Size, 0, 0, info.pixID]

ENDIF
END
ENDCASE
Widget_Control, event.top, Set_UValue=info, /No_Copy
END

```

```

PRO Histo_GUI_Drawing_Colors, event
; This event handler changes the drawing colors of the graphic display.

Widget_Control, event.top, Get_UValue=info, /No_Copy
; Which color are we changing? The button UVALUE will tell us.

Widget_Control, event.id, Get_UValue=buttonUValue
; Change it by calling the modal dialog PickColorName.

CASE buttonUValue OF
'ANNOTATION': BEGIN
colorname = PickColorName(info.axisColorName, $
Cancel=cancelled, Group_Leader=event.top, $
Title='Select Annotation Color', $
Index=!D.Table_Size-2, Bottom=!D.Table_Size-21)
IF NOT cancelled THEN info.axisColorName = colorname
END
'DATA': BEGIN
colorname = PickColorName(info.dataColorName, $
Cancel=cancelled, Group_Leader=event.top, $
Title='Select Data Color', $
Index=!D.Table_Size-3, Bottom=!D.Table_Size-21)
IF NOT cancelled THEN info.dataColorName = colorname
END
'BACKGROUND': BEGIN
colorname = PickColorName(info.backColorName, $
Cancel=cancelled, Group_Leader=event.top, $
Title='Select Background Color', $
Index=!D.Table_Size-4, Bottom=!D.Table_Size-21)
IF NOT cancelled THEN info.backColorName = colorname
END
ENDCASE
; Retrieve the new color table. The keyboard focus events will redraw the
; graphics display.

TVLCT, r, g, b, /Get
info.r = r
info.g = g
info.b = b

Widget_Control, event.top, Set_UValue=info, /No_Copy
END

```

```

PRO Histo_GUI_Undo, event
; This event handler responds to the UNDO button.
Widget_Control, event.top, Get_UValue=info, /No_Copy
; Switch the process and undo images.
temp = *info.process
*info.process = *info.undo
*info.undo = temp
; Switch the UNDO/REDO button values.
Widget_Control, event.id, Get_Value=theValue, Get_UValue=theUValue
Widget_Control, event.id, Set_Value=theUValue, Set_UValue=theValue
; Make the pixmap window the current graphics window.
WSet, info.pixID
; Draw the graphics.
HistoImage, *info.process, $
AxisColorName=info.axisColorName, $
BackColorName=info.backcolorName, $
Binsize=info.binsize, $
DataColorName=info.datacolorName, $
_Extra=*info.extra, $
Max_Value=info.max_value, $
NoLoadCT=1, $
XScale=info.xscale, $
YScale=info.yscale
WSet, info.wid
Device, Copy=[0, 0, !D.X_Size, !D.Y_Size, 0, 0, info.pixID]
Widget_Control, event.top, Set_UValue=info, /No_Copy
END

```

```

PRO Histo_GUI_Processing, event
; This event handler responds to image processing buttons.
Widget_Control, event.top, Get_UValue=info, /No_Copy
; Set the undo image to be the current process image.
*info.undo = *info.process
; Set the undo button to UNDO and make it sensitive.
Widget_Control, info.undoID, Set_Value='Undo', Set_UValue='Redo', Sensitive=1
; What kind of processing do you need?
Widget_Control, event.id, Get_Value=buttonValue
CASE StrUpCase(buttonValue) OF
'MEDIAN SMOOTH': *info.process = Median(*info.process, 5)
'BOXCAR SMOOTH': *info.process = Smooth(*info.process, 7, /Edge_Truncate)
'SOBEL': *info.process = Sobel(*info.process)
'UNSHARP MASKING': *info.process = Smooth(*info.process, 7) - *info.process
'ORIGINAL IMAGE': *info.process = *info.image
ENDCASE
; Display the new image and its histogram.
WSet, info.pixID
; Draw the graphics.
HistoImage, *info.process, $
AxisColorName=info.axisColorName, $
BackColorName=info.backcolorName, $

```

```

Binsize=info.binsize, $
DataColorName=info.datacolorName, $
_Extra=*info.extra, $
Max_Value=info.max_value, $
NoLoadCT=1, $
XScale=info.xscale, $
YScale=info.yscale
Widget_Control, event.top, Set_UValue=info, /No_Copy
WSet, info.wid
Device, Copy=[0, 0, !D.X_Size, !D.Y_Size, 0, 0, info.pixID]
END

```

```

PRO Histo_GUI_Quit, event
Widget_Control, event.top, /Destroy
END

```

```

PRO Histo_GUI_TLB_Events, event
; This event handler responds to keyboard focus and resize events.
thisEvent = Tag_Names(event, /Structure_Name)
IF thisEvent EQ 'WIDGET_BASE' THEN BEGIN
    ; Get the info structure and copy it here.
    Widget_Control, event.top, Get_UValue=info, /No_Copy
    ; Resize the draw widget.
    Widget_Control, info.drawID, Draw_XSize=event.x, Draw_YSize=event.y
    ; Delete the current pixmap and make another the proper size.
    WDelete, info.pixID
    Window, /Free, /Pixmap, XSize=event.x, YSize=event.y
    info.pixID = !D.Window
    ; Draw the graphics.
    HistoImage, *info.process, $
        AxisColorName=info.axisColorName, $
        BackColorName=info.backcolorName, $
        Binsize=info.binsize, $
        DataColorName=info.datacolorName, $
        _Extra=*info.extra, $
        Max_Value=info.max_value, $
        NoLoadCT=1, $
        XScale=info.xscale, $
        YScale=info.yscale
    WSet, info.wid
    Device, Copy=[0, 0, !D.X_Size, !D.Y_Size, 0, 0, info.pixID]
    ; Put the info structure back in its storage location.
    Widget_Control, event.top, Set_UValue=info, /No_Copy
ENDIF
IF thisEvent EQ 'WIDGET_KBRD_FOCUS' THEN BEGIN
    ; If losing keyboard focus, do nothing and RETURN.
    IF event.enter EQ 0 THEN RETURN
    ; Get the info structure and copy it here.
    Widget_Control, event.top, Get_UValue=info, /No_Copy

```

Appendix C: IDL Program Code

```
; Load the program's colors.  
TVLCT, info.r, info.g, info.b  
; If this is other than 8-bit display, redraw graphic.  
Device, Get_Visual_Depth=theDepth  
IF theDepth GT 8 THEN BEGIN  
WSet, info.pixID  
HistoImage, *info.process, $  
AxisColorName=info.axisColorName, $  
BackColorName=info.backcolorName, $  
Binsize=info.binsize, $  
DataColorName=info.datacolorName, $  
_Extra=*info.extra, $  
Max_Value=info.max_value, $  
NoLoadCT=1, $  
XScale=info.xscale, $  
YScale=info.yscale  
WSet, info.wid  
Device, Copy=[0, 0, !D.X_Size, !D.Y_Size, 0, 0, info.pixID]  
ENDIF  
;  
; Put the info structure back in its storage location.  
Widget_Control, event.top, Set_UValue=info, /No_Copy  
ENDIF  
END
```

```
PRO Histo_GUI_Cleanup, tlb  
;  
; The purpose of this procedure is to clean up pointers,  
; objects, pixmaps, and other things in our program that  
; use memory. This procedure is called when the top-level  
; base widget is destroyed.  
Widget_Control, tlb, Get_UValue=info, /No_Copy  
IF N_Elements(info) EQ 0 THEN RETURN  
;  
; Free the pointers.  
Ptr_Free, info.image  
Ptr_Free, info.extra  
WDelete, info.pixID  
Ptr_Free, info.process  
Ptr_Free, info.undo  
END
```

```
PRO Histo_GUI, $  
image, $ ; The program name.  
AxisColorName=axisColorName, $ ; The image data.  
BackColorName=backcolorName, $ ; The axis color.  
Binsize=binsize, $ ; The background color.  
ColorTable=colortable, $ ; The histogram bin size.  
DataColorName=datacolorName, $ ; The colortable index to load.  
_Extra=extra, $ ; The data color.  
Group_Leader=group_leader, $ ; For passing extra keywords.  
Max_Value=max_value, $ ; The group leader for the TLB.  
Title=title, $ ; The maximum value of the histogram plot.  
XScale=xscale, $ ; The program window title.  
YScale=yscale ; The scale for the X axis of the image.  
YScale=yscale ; The scale for the Y axis of the image.  
;  
; Catch any error in the Histo_GUI program.
```

```

Catch, theError
IF theError NE 0 THEN BEGIN
  Catch, /Cancel
  ok = Error_Message(Traceback=1)
  RETURN
ENDIF

; Check for positional parameter. Define if necessary.
; Make sure it is correct size.

IF N_Elements(image) EQ 0 THEN image = LoadData(7)
ndim = Size(image, /N_Dimensions)
IF ndim NE 2 THEN Message, '2D Image Variable Required.', /NoName

; Check for histogram keywords.

IF N_Elements(binsize) EQ 0 THEN BEGIN
  range = Max(image) - Min(image)
  binsize = 2.0 > (range / 128.0)
ENDIF

IF N_Elements(max_value) EQ 0 THEN max_value = 5000.0
IF N_Elements(title) EQ 0 THEN title = 'Histo_GUI Program'

; Check for image scale parameters.

s = Size(image, /Dimensions)
IF N_Elements(xscale) EQ 0 THEN xscale = [0, s[0]]
IF N_Elements(xscale) NE 2 THEN Message, 'XSCALE must be 2-element array', /NoName
IF N_Elements(yscale) EQ 0 THEN yscale = [0, s[1]]
IF N_Elements(yscale) NE 2 THEN Message, 'YSCALE must be 2-element array', /NoName

; Check for color keywords.

IF N_Elements(dataColorName) EQ 0 THEN dataColorName = "Red"
IF N_Elements(axisColorName) EQ 0 THEN axisColorName = "Navy"
IF N_Elements(backcolorName) EQ 0 THEN backcolorName = "White"
IF N_Elements(colortable) EQ 0 THEN colortable = 4
colortable = 0 > colortable < 40
imagecolors = !D.Table_Size-4

; Define the TLB. The TLB should be resizeable and it should have a menu bar.

tlb = Widget_Base(Column=1, /TLB_Size_Events, Title=title, MBar=menubarID)

; Define the File pull-down menu.

fileID = Widget_Button(menubarID, Value='File')

; Define an Open button.

openID = Widget_Button(fileID, Value='Open...', Event_Pro='Histo_GUI_Open_Image')

; Define the Print pull-down menu.

printID = Widget_Button(fileID, Value='Print', Event_Pro='Histo_GUI_Print', /Menu)
button = Widget_Button(printID, Value='Portrait Mode')
button = Widget_Button(printID, Value='Landscape Mode')

; Define the Save As pull-down menu.

saveID = Widget_Button(fileID, Value='Save As', Event_Pro='Histo_GUI_File_Output', /Menu)
button = Widget_Button(saveID, Value='GIF File', UValue='.gif')
button = Widget_Button(saveID, Value='JPEG File', UValue='.jpg')
button = Widget_Button(saveID, Value='TIFF File', UValue='.tif')
button = Widget_Button(saveID, Value='PostScript File', UValue='.ps')

; Define the Quit button.

quitID = Widget_Button(fileID, Value='Quit', Event_Pro='Histo_GUI_Quit', /Separator)

; Define the Processing pull-down menu.

processID = Widget_Button(menubarID, Value='Processing', $
  Event_Pro='Histo_GUI_Processing', /Menu)
smoothID = Widget_Button(processID, Value='Smoothing', /Menu)
button = Widget_Button(smoothID, Value='Median Smooth')
button = Widget_Button(smoothID, Value='Boxcar Smooth')

```

Appendix C: IDL Program Code

```

edgeID = Widget_Button(processID, Value='Edge Enhance', /Menu)
button = Widget_Button(edgeID, Value='Sobel')
button = Widget_Button(edgeID, Value='Unsharp Masking')
button = Widget_Button(processID, Value='Original Image')
undoID = Widget_Button(processID, Value='Undo', UValue='Redo', $
    Event_Pro='Histo_GUI_Undo', Sensitive=0, /Separator)
; Define the Colors pull-down menu.

colorsID = Widget_Button(menubarID, Value='Colors')
button = Widget_Button(colorsID, Value='Image Colors', Event_Pro='Histo_GUI_Image_Colors')
drawColorsID = Widget_Button(colorsID, Value='Drawing Colors',
    Event_Pro='Histo_GUI_Drawing_Colors', /Menu)
button = Widget_Button(drawColorsID, Value='Data Color', UValue='DATA')
button = Widget_Button(drawColorsID, Value='Background Color', UValue='BACKGROUND')
button = Widget_Button(drawColorsID, Value='Annotation Color', UValue='ANNOTATION')

; Define the draw widget.

drawID = Widget_Draw(tlb, XSize=400, YSize=400)
; Realize the widget hierarchy.

Widget_Control, tlb, /Realize
; Get the window index number of the draw widget window. Make it active.

Widget_Control, drawID, Get_Value=wid
; Create a pixmap for double buffering of graphics display output.
; Double buffering will result in smoother graphically display.

Window, /Free, /Pixmap, XSize=400, YSize=400
pixID = !D.Window
; Draw the graphic display.

HistoImage, image, $
AxisColorName=axisColorName, $
BackColorName=backcolorName, $
Binsize=binsize, $
ColorTable=colortable, $
DataColorName=datacolorName, $
_Extra=extra, $
ImageColors=imagecolors, $
Max_Value=max_value, $
XScale=xscale, $
YScale=yscale
; Copy the graphical output from the pixmap to the display window.

WSet, wid
Device, Copy=[0, 0, !D.X_Size, !D.Y_Size, 0, 0, pixID]
; Obtain the current RGB color vectors.

TVLCT, r, g, b, /Get
; Create the info structure with the information
; required to run the program.

info = { image:Ptr_New(image), $ ; A pointer to the image data.
        process:Ptr_New(image), $ ; A pointer to the process image.
        undo:Ptr_New(image), $ ; A pointer to last processed image.
        undoID:undoID, $ ; The identifier of the UNDO button.
        axisColorName:axisColorName, $ ; The name of the axis color.
        backColorName:backcolorName, $ ; The name of the background color.
        binsize:binsize, $ ; The histogram bin size.
        dataColorName:datacolorName, $ ; The name of the data color.
        imageColors:imagecolors, $ ; The number of colors used for image.
        max_value:max_value, $ ; The maximum value of histogram plot.
        xscale:xscale, $ ; The X scale of the image axis.
        yscale:yscale, $ ; The Y scale of the image axis.
        title:title, $ ; The window title.
        extra:Ptr_New(extra), $ ; A pointer to "extra" keywords.

```

```

        r:r, $                                ; The R color vector.
        g:g, $                                ; The G color vector.
        b:b, $                                ; The B color vector.
        drawID:drawID, $                      ; The identifier of the draw widget.
        wid:wid, $                            ; The index number of graphics window.
        pixID:pixID $                        ; The index number of the pixmap window.
    }

; Store the info structure in the user value of the TLB. Turn keyboard
; focus events on.

Widget_Control, tlb, Set_UValue=info, /No_Copy, /KBRD_Focus_Events
; Set up the event loop. Register the program with the window manager.

XManager, 'histo_gui', tlb, Event_Handler='Histo_GUI_TLB_Events', $
    /No_Block, Cleanup='Histo_GUI_Cleanup', Group_Leader=group_leader
END

```

HistoImage__Define Object Program

This is an example of a graphics display program written as an object. It is described in detail in “Creating a New Object Class” on page 353. The source code for this program can be downloaded from the *Coyote’s Guide to IDL Programming* anonymous ftp site. The URL is:

ftp://ftp.dfanning.com/pub/dfanning/outgoing/coyote2nd/histoimage__define.pro

```

PRO HistoImage::GetProperty, $
    Binsize=binsize, $                      ; The bin size of the histogram.
    DataColor=datacolor, $                  ; The data color of the histogram.
    Max_Value=max_value, $                ; The maximum value of the histogram plot.
    _Ref_Extra=extra                         ; Extra keywords sent to the BoxImage superclass.

; This method sets the properties of the object. Extra keywords
; are passed along to and retrieved from the BoxImage superclass object.

; Error handling.

Catch, theError
IF theError NE 0 THEN BEGIN
    Catch, /Cancel
    ok = Error_Message(!Error_State.Msg + ' Returning...', $
        Traceback=1, /Error)
    RETURN
ENDIF

; Set properties if keyword is present.

IF Arg_Present(binsize) NE 0 THEN binsize = self.binsize
IF Arg_Present(datacolor) NE 0 THEN datacolor = self.datacolor
IF Arg_Present(max_value) NE 0 THEN max_value = self.max_value

; Pass extra keywords along to the BoxImage superclass.

self->BoxImage::GetProperty, _Extra=extra
END

```

```

PRO HistoImage::SetProperty, $
    Binsize=binsize, $                      ; The bin size of the histogram.
    DataColor=datacolor, $                  ; The data color of the histogram.
    Max_Value=max_value, $                ; The maximum value of the histogram plot.
    _Extra=extra                           ; Extra keywords sent to the BoxImage superclass.

```

Appendix C: IDL Program Code

```
; This method sets the properties of the object. Extra keywords  
; are passed along to and set properties in the BoxImage superclass object.  
;  
; Error handling.  
  
Catch, theError  
IF theError NE 0 THEN BEGIN  
    Catch, /Cancel  
    ok = Error_Message(!Error_State.Msg + ' Returning...', $  
        Traceback=1, /Error)  
    RETURN  
ENDIF  
  
; Set properties if keyword is present.  
  
IF N_Elements(binsize) NE 0 THEN self.binsize = binsize  
IF N_Elements(datacolor) NE 0 THEN self.datacolor = datacolor  
IF N_Elements(max_value) NE 0 THEN self.max_value = max_value  
  
; Pass extra keywords along to the BoxImage superclass.  
  
self->BoxImage::SetProperty, _Extra=extra  
END
```

```
PRO HistoImage::DataColor, Draw=draw, _Extra=extra  
;  
; This method changes the data color.  
  
thisColorName = PickColorName(self.datacolor, Cancel=cancelled, $  
    _Extra=extra, Title='Data Color')  
IF cancelled THEN RETURN  
  
self.datacolor = thisColorName  
  
; Redraw the image if needed.  
  
IF Keyword_Set(draw) THEN self->Draw  
END
```

```
PRO HistoImage::Draw, $  
Font=font, $                                ; Type of font wanted for output.  
BMP=bmp, $                                    ; Write to BMP file if set.  
GIF=gif, $                                    ; Write to GIF file if set.  
JPEG=jpeg, $                                    ; Write to JPEG file if set.  
PICT=pict, $                                    ; Write to PICT file if set.  
PNG=png, $                                    ; Write to PNG file if set.  
TIFF=Tiff, $                                    ; Write to TIFF file if set.  
PostScript=postscript, $                        ; Write to PostScript file if set.  
PS=ps, $                                         ; Write to PostScript file if set.  
Printer=printer, $                            ; Write directly to Printer if set.  
    _Extra=extra                                ; Extra keywords (e.g., for PSConfig).  
  
; This method draws the graphics display. Polymorphism is illustrated  
; the the draw method working in a variety of devices.  
;  
; Error handling.  
  
Catch, theError  
IF theError NE 0 THEN BEGIN  
    Catch, /Cancel  
    ok = Error_Message(!Error_State.Msg + ' Returning...', $  
        Traceback=1, /Error)  
    RETURN  
ENDIF  
  
; Check keywords.
```

```

IF N_Elements(font) EQ 0 THEN font = !P.Font
; Special output?

output = ""
IF Keyword_Set(bmp) THEN output = 'BMP'
IF Keyword_Set(gif) THEN output = 'GIF'
IF Keyword_Set(jpeg) THEN output = 'JPEG'
IF Keyword_Set(pict) THEN output = 'PICT'
IF Keyword_Set(png) THEN output = 'PNG'
IF Keyword_Set(tiff) THEN output = 'TIFF'
IF Keyword_Set(postscript) THEN output = 'PS'
IF Keyword_Set(ps) THEN output = 'PS'
IF Keyword_Set(printer) THEN output = 'PRINTER'
IF output NE "" THEN thisDevice = !D.Name

; Setup based on type of output.

CASE output OF

": BEGIN
annotateColor = GetColor(self.annotateColor, !D.Table_Size-2)
backColor = GetColor(self.backColor, !D.Table_Size-3)
dataColor = GetColor(self.dataColor, !D.Table_Size-4)
TVLCT, *self.r, *self.g, *self.b
END

"PS": BEGIN
keywords = PSConfig(Color=1, Filename='histoimage.ps', $
    _Extra=extra, Cancel=cancelled)
IF cancelled THEN RETURN ELSE keywords.color = 1
Set_Plot, 'PS'
Device, _Extra=keywords
annotateColor = GetColor('Navy', !D.Table_Size-2)
backColor = GetColor('White', !D.Table_Size-3)
dataColor = GetColor('Black', !D.Table_Size-4)
TVLCT, *self.r, *self.g, *self.b
END

"PRINTER": BEGIN
ok = Dialog_PrinterSetup()
IF NOT ok THEN RETURN
keywords = PSWindow(/Printer, /Landscape, Fudge=0.25, _Extra=extra)
annotateColor = GetColor('Black', !D.Table_Size-2)
backColor = GetColor('Charcoal', !D.Table_Size-3)
dataColor = GetColor('Black', !D.Table_Size-4)
TVLCT, *self.r, *self.g, *self.b
Set_Plot, 'PRINTER', /Copy
Device, Landscape=1
Device, _Extra=extra
thisThickness = !P.Thick
thisFont = !P.Font
font = 1
!P.Thick = 2
END

ELSE: BEGIN
ncolors = !D.Table_Size
Set_Plot, 'Z'
Device, Set_Resolution=[500, 500], Set_Colors=ncolors, _Extra=extra
Erase
annotateColor = GetColor(self.annotateColor, ncolors-2)
backColor = GetColor(self.backColor, ncolors-3)
dataColor = GetColor(self.dataColor, ncolors-4)
TVLCT, *self.r, *self.g, *self.b
END

ENDCASE

; Calculate the position of the image, color bar,
; and histogram plots in the window.

```

Appendix C: IDL Program Code

```

IF self.vertical THEN BEGIN
  p = self.position
  length = p[2] - p[0]
  imgpos = [p[0], p[1], (p[0] + (0.75*length)), p[3]-(length*0.350)]
  cbpos = [p[2]-0.05, p[1], p[2], p[3]-(length*0.35)]
  hpos = [p[0], imgpos[3]+0.1, p[2], p[3]]
ENDIF ELSE BEGIN
  p = self.position
  height = p[3] - p[1]
  imgpos = [p[0], p[1], p[2], p[1]+ 0.4*height]
  cbpos = [p[0], imgpos[3]+height*0.1, p[2], imgpos[3]+height*0.15]
  hpos = [p[0], cbpos[3]+height*0.125, p[2], p[3]]
ENDELSE
; Calculate appropriate character size for plots.

IF output EQ 'PRINTER' THEN thisCharsize = 1.25 ELSE $
  thisCharsize = Str_Size('A Sample String', 0.20)

; Calculate the histogram.

histdata = Histogram(*self.process, Binsize=self.binsize, $
  Max=Max(*self.process), Min=Min(*self.process))

; Have to fudge the bins and histdata variables to get the
; histogram plot to make sense.

npts = N_Elements(histdata)
halfbinsize = self.binsize / 2.0
bins = Findgen(N_Elements(histdata)) * self.binsize + Min(*self.process)
binsToPlot = [bins[0], bins + halfbinsize, bins[npts-1] + self.binsize]
histdataToPlot = [histdata[0], histdata, histdata[npts-1]]
xrange = [Min(binsToPlot), Max(binsToPlot)]

; Plot the histogram of the display image. Axes first.

Plot, binsToPlot, histdataToPlot, $ ; The fudged histogram and bin data.
  Background=backcolor, $ ; The background color of the display.
  CharSize=thisCharsize, $ ; The character size, as determined by Str_Size.
  Color=annotateColor, $ ; The color of the axes.
  Font=font, $ ; The font type.
  Max_Value=self.max_value, $ ; The maximum value of the plot.
  NoData=1, $ ; Draw the axes only. No data.
  Position=hpos, $ ; The position of the plot in the window.
  Title='Image Histogram', $ ; The title of the plot.
  XRange=xrange, $ ; The X data range.
  XStyle=1, $ ; Exact axis scaling. No autoscaled axes.
  XTickFormat='(I6)', $ ; The format of the X axis annotations.
  XTitle='Image Value', $ ; The title of the X axis.
  YMinor=1, $ ; One minor tick mark on X axis.
  YRange=[0,self.max_value], $ ; The Y data range.
  YStyle=1, $ ; Exact axis scaling. No autoscaled axes.
  YTICKFORMAT='(I6)', $ ; The format of the Y axis annotations.
  YTitle='Pixel Density', $ ; The title of the Y axis.
  _Extra=*self.extra ; Pass any extra PLOT keywords.

; Overplot the histogram data in the data color.

OPlot, binsToPlot, histdataToPlot, PSym=10, Color=dataColor
; Make histogram boxes by drawing lines in data color.

FOR j=1L,N_Elements(bins)-2 DO BEGIN
  PlotS, [bins[j], bins[j]], [!Y.CRange[0], histdata[j] < self.max_value], $
    Color=dataColor, _Extra=*self.extra
ENDFOR
; Draw the image and color bar.

TVImage, BytScl(*self.process, Top=self.ncolors-1), $
  Position=imgpos, _Extra=*self.extra
Plot, self.xscale, self.yscale, XStyle=1, YStyle=1, $
  XTitle=self.xtitle, YTitle=self.ytitle, Color=annotateColor, $

```

```

Position=imgpos, /NoErase, /NoData, Ticklen=-0.025, _Extra=*self.extra, $
CharSize=thisCharSize, Font=font
Colorbar, Range=[Min(*self.process), Max(*self.process)], Divisions=8, $
_Extra=*self.extra, Color=annotateColor, Position=cbpos, Ticklen=-0.2, $
Vertical=self.vertical, NColors=self.ncolors, CharSize=thisCharSize, Font=font
; Do you need file output? Get a screen dump and write the file.

CASE output OF
  "BMP": image = TVRead(/BMP,  Filename='histoimage')
  "GIF": image = TVRead(/GIF,  Filename='histoimage')
  "JPEG": image = TVRead(/JPEG,  Filename='histoimage')
  "PNG": image = TVRead(/PNG,  Filename='histoimage')
  "PICT": image = TVRead(/PICT,  Filename='histoimage')
  "TIFF": image = TVRead(/TIFF,  Filename='histoimage')
  "PS": Device, Close_File=1
  "PRINTER": BEGIN
    Device, Close_Document=1
    !P.Thick = thisThickness
  END
  ELSE:
ENDCASE
IF output NE "" THEN Set_Plot, thisDevice
END

```

```

FUNCTION HistoImage::Init, $
  image, $
  Binsize=binsize, $           ; The bin size of the histogram.
  DataColor=datacolor, $       ; The data color.
  Max_Value=max_value, $       ; The maximum value of the histogram plot.
  _Extra=extra                 ; Holds extra keywords.

; The initialization routine for the object. Create the
; particular instance of the object class.

; Error handling.

Catch, theError
IF theError NE 0 THEN BEGIN
  Catch, /Cancel
  ok = Error_Message(!Error_State.Msg + ' Returning...', $
    Traceback=1, /Error)
  RETURN, 0
ENDIF

; Check for positional parameter. Define if necessary.

IF N_Elements(image) EQ 0 THEN image = LoadData(7)
ndims = Size(image, /N_Dimensions)
IF ndims NE 2 THEN Message, 'Image must be 2D array.', /NoName

; Check for keyword parameters.

IF N_Elements(datacolor) EQ 0 THEN datacolor = "RED"
IF N_Elements(max_value) EQ 0 THEN max_value = 5000.0
IF N_Elements(binsize) EQ 0 THEN BEGIN
  range = Max(image) - Min(image)
  binsize = 2.0 > (range / 128.0)
ENDIF

; Initialize the BoxImage superclass object.

IF NOT self->BoxImage::Init(image, _Extra=extra, NColors=!D.Table_Size-4) THEN RETURN, 0
; Populate the rest of the self object.

self.max_value = max_value
self.datacolor = datacolor
self.binsize = binsize

```

Appendix C: IDL Program Code

```
RETURN, 1  
END
```

```
PRO HistoImage__Define  
; The definition of the HISTOIMAGE object class.  
struct = { HISTOIMAGE, $ ; The HISTOIMAGE object class.  
          INHERITS BoxImage, $ ; Inherit the BoxImage object class.  
          binsize: 0.0, $ ; The histogram bin size.  
          max_value: 0.0, $ ; The maximum value of the histogram plot.  
          datacolor: "" $ ; The data color name.  
        }  
END
```

OpenImage Program

This is an example of a modal or blocking dialog form widget program. Additional information about this program can be found in “Creating a Modal Dialog Form Widget” on page 325. The source code for this program can be downloaded from the *coyote* anonymous ftp site. The URL is:

```
ftp://ftp.dfanning.com/pub/dfanning/outgoing/coyote2nd/openimage.pro
```

```
Pro OpenImage_BrowseFiles, event  
filename = Dialog_Pickfile(Filter='*.dat')  
IF filename EQ "" THEN RETURN  
    ; Update file name text widget.  
Widget_Control, event.top, Get_UValue=info, /No_Copy  
info.fileID->Set_Value, filename  
Widget_Control, event.top, Set_UValue=info, /No_Copy  
END
```

```
Pro OpenImage_Events, event  
    ; Error handling.  
Catch, theError  
IF theError NE 0 THEN BEGIN  
    Catch, /Cancel  
    IF !Error_State.Code EQ -167 THEN BEGIN  
        ok = Error_Message('A required value is undefined.')  
    ENDIF ELSE BEGIN  
        ok= Error_Message()  
    ENDELSE  
    IF N_Elements(info) NE 0 THEN $  
        Widget_Control, event.top, Set_UValue=info, /No_Copy  
    RETURN  
ENDIF  
    ; What kind of event is this? We only want to handle button events  
    ; from our ACCEPT or CANCEL buttons. Other events fall through.  
eventName = Tag_Names(event, /Structure_Name)  
IF eventName NE 'WIDGET_BUTTON' THEN RETURN  
    ; Get the info structure out of the top-level base
```

```

Widget_Control, event.top, Get_UValue=info, /No_Copy
; Which button was selected?
Widget_Control, event.id, Get_Value=buttonValue
CASE buttonValue OF
  'Cancel' : Widget_Control, event.top, /Destroy
  'Accept' : BEGIN
    ; Fill out the file data structure with information
    ; collected from the form. Be sure to get just the
    ; *first* filename, since values from text widgets are
    ; always string arrays. Set the CANCEL flag correctly.
    filename = info.fileID->Get_Value()
    filename = filename[0]
    xsize = info.xsizeID->Get_Value()
    ysize = info.ysizeID->Get_Value()
    ; Preliminary checks of the fileInfo information.
    ; Does the file really exist?
    dummy = Findfile(filename, Count=theCount)
    IF theCount EQ 0 THEN $
      Message, 'Requested file cannot be found. Check spelling.', /NoName
      ; Are the file sizes positive?
    IF xsize LE 0 OR ysize LE 0 THEN $
      Message, 'File sizes must be positive numbers.', /NoName
      ; If it checks out, set the pointer information.
    (*info.ptr).filename = filename
    (*info.ptr).xsize = xsize
    (*info.ptr).ysize = ysize
    (*info.ptr).cancel = 0
    ; Destroy the widget program
    Widget_Control, event.top, /Destroy
  END
ENDCASE
END

```

```

Function OpenImage, $
  Filename=filename, $          ; Initial name of file to open.
  Group_Leader=group_leader, $   ; Group leader of this program.
  XSize=xsize, $                ; Initial X size of file to open.
  YSize=ysize, $                ; Initial Y size of file to open.
  Cancel=cancel                 ; An output cancel flag.

; This is a pop-up dialog widget to collect the filename and
; file sizes from the user. The widget is a modal or blocking
; widget. The function result is the image that is read from
; the file.

; The Cancel field indicates whether the user clicked the CANCEL
; button (result.cancel=1) or the ACCEPT button (result.cancel=0).

On_Error, 2 ; Return to caller.

; Check parameters and keywords.

IF N_Elements(filename) EQ 0 THEN $
  filename=Filepath(SubDirectory=['examples','data'],'ctscan.dat')
IF N_Elements(xsize) EQ 0 THEN xsize = 256
IF N_Elements(ysize) EQ 0 THEN ysize = 256
; Create a top-level base. Must have a Group Leader defined

```

Appendix C: IDL Program Code

```
; for Modal operation. If this widget is NOT modal, then it
; should only be called from the IDL command line as a blocking
; widget.

IF N_Elements(group_leader) NE 0 THEN $
  tlb = Widget_Base(Column=1, Title='Enter File Information...', /Modal, $
    Group_Leader=group_leader, /Floating, /Base_Align_Center) ELSE $
  tlb = Widget_Base(Column=1, Title='Enter File Information...', $
    /Base_Align_Center)

; Make sub-bases.

subbase = Widget_Base(tlb, Column=1, Frame=1)
filebase = Widget_Base(subbase, Row=1)

; Create widgets for filename. Set text widget size appropriately.

filesize = StrLen(filename) * 1.25
fileID = FSC_InputField(filebase, Title='Filename:', Value=filename, $
  XSize=filesize, LabelSize=50, /StringValue)
browseID = Widget_Button(filebase, Value='Browse', Event_Pro='OpenImage_BrowseFiles')
xsizeID = FSC_InputField(subbase, Title='X Size:', $
  Value=xsize, /IntegerValue, LabelSize=50, Digits=4)
ysizeID = FSC_InputField(subbase, Title='Y Size:', $
  Value=ysize, /IntegerValue, LabelSize=50, Digits=4)

; Set up Tabing between fields.

fileID->SetTabNext, xsizeID->GetTextID()
xsizeID->SetTabNext, ysizeID->GetTextID()
ysizeID->SetTabNext, fileID->GetTextID()

; Make a button base with frame to hold CANCEL and ACCEPT buttons.

butbase = Widget_Base(tlb, Row=1)
cancel = Widget_Button(butbase, Value='Cancel')
accept = Widget_Button(butbase, Value='Accept')

; Center the program on the display

screenSize = Get_Screen_Size()
geom = Widget_Info(tlb, /Geometry)
Widget_Control, tlb, $
  XOffset = (screenSize[0] / 2) - (geom.scr_xsize / 2), $
  YOffset = (screenSize[1] / 2) - (geom.scr_ysize / 2)

; Realize top-level base and all of its children.

Widget_Control, tlb, /Realize

; Create a pointer. This will point to the location where the
; information collected from the user will be stored. You must
; store it external to the widget program, since the program
; will be destroyed no matter which button is selected. Fill the
; pointer with NULL values.

ptr = Ptr_New({Filename:'', Cancel:1, XSize:0, YSize:0})

; Create info structure to hold information needed in event handler.

info = { fileID:fileID, $           ; Identifier of widget holding filename.
         xsizeID:xsizeID, $     ; Identifier of widget holding xsize.
         ysizeID:ysizeID, $     ; Identifier of widget holding ysize.
         ptr:ptr }             ; Pointer to file information storage location.

; Store the info structure in the top-level base

Widget_Control, tlb, Set_UValue=info, /No_Copy

; Register the program, set up event loop. Make this program a
; blocking widget. This will allow the program to also be called
; from IDL command line without a GROUP_LEADER parameter. The program
; blocks here until the entire program is destroyed.

XManager, 'openimage', tlb, Event_Handler='OpenImage_Events'

; OK, widget is destroyed. Go get the file information in the pointer
```

```

; location, free up pointer memory, and return the file information.

fileInfo = *ptr
Ptr_Free, ptr

; Set the Cancel flag.

cancel = fileInfo.cancel

; Return the file information.

RETURN, fileInfo
END

```

ReadImage Program

This is an example of a non-modal dialog form widget program. Additional information about this program can be found in “Creating a Non-Modal Widget Dialog” on page 340. The source code for this program can be downloaded from the *coyote* anonymous ftp site. The URL is:

<ftp://ftp.dfanning.com/pub/dfanning/outgoing/coyote2nd/readimage.pro>

```

Pro ReadImage_BrowseFiles, event
filename = Dialog_Pickfile(Filter='*.dat')
IF filename EQ "" THEN RETURN
    ; Update file name text widget.

Widget_Control, event.top, Get_UValue=info, /No_Copy
info.fileID->Set_Value, filename
Widget_Control, event.top, Set_UValue=info, /No_Copy
END

```

```

Pro ReadImage_Events, event
    ; Error handling.

Catch, theError
IF theError NE 0 THEN BEGIN
    Catch, /Cancel
    IF !Error_State.Code EQ -167 THEN BEGIN
        ok = Error_Message('A required value is undefined.')
    ENDIF ELSE BEGIN
        ok= Error_Message()
    ENDELSE
    IF N_Elements(info) NE 0 THEN $
        Widget_Control, event.top, Set_UValue=info, /No_Copy
    RETURN
ENDIF
    ; What kind of event is this? We only want to handle button events
    ; from our ACCEPT or CANCEL buttons. Other events fall through.

eventName = Tag_Names(event, /Structure_Name)
IF eventName NE 'WIDGET_BUTTON' THEN RETURN
    ; Get the info structure out of the top-level base
Widget_Control, event.top, Get_UValue=info, /No_Copy
    ; Which button was selected?

Widget_Control, event.id, Get_Value=buttonValue
CASE buttonValue OF

```

Appendix C: IDL Program Code

```

'Dismiss' : Widget_Control, event.top, /Destroy
'Apply' : BEGIN
    ; Fill out the file data structure with information
    ; collected from the form. Be sure to get just the
    ; *first* filename, since values from text widgets are
    ; always string arrays. Set the CANCEL flag correctly.

    filename = info.fileID->Get_Value()
    filename = filename[0]
    xsize = info.xsizeID->Get_Value()
    ysize = info.ysizeID->Get_Value()

    ; Preliminary checks of the fileInfo information.
    ; Does the file really exist?

    dummy = Findfile(filename, Count=theCount)
    IF theCount EQ 0 THEN $
        Message, 'Requested file cannot be found. Check spelling.', /NoName
    ; Are the file sizes positive?

    IF xsize LE 0 OR ysize LE 0 THEN $
        Message, 'File sizes must be positive numbers.', /NoName
    ; If it checks out, send an event.

    s = Size(info.notifyIDs)
    IF s[0] EQ 1 THEN count = 0 ELSE count = s[2] - 1
    FOR j=0,count DO BEGIN
        ; Create a fileInfo event.

        fileInfo = { READIMAGE_EVENT, $
                     ID:info.notifyIDs[0,j], $
                     Top:info.notifyIDs[1,j], $
                     Handler:0L, $
                     Filename:filename, $
                     XSize:xsize, $
                     YSize:ysize }

        IF Widget_Info(info.notifyIDs[0,j], /Valid_ID) THEN $
            Widget_Control, info.notifyIDs[0,j], Send_Event=fileInfo
    ENDFOR
    Widget_Control, event.top, Set_UValue=info, /No_Copy
END
ENDCASE
END

```

```

PRO ReadImage, $
    notifyIDs, $                      ; A vector of widgets and their TLBs to notify.
    Filename=filename, $                ; Initial name of file to open.
    Group_Leader=group_leader, $        ; Group leader of this program.
    XSize=xsize, $                     ; Initial X size of file to open.
    YSize=ysize                         ; Initial Y size of file to open.

    ; This is a pop-up dialog widget to collect the filename and
    ; file sizes from the user. The widget is non-modal.
    ; Only one READIMAGE program at a time.

IF XRegistered('readimage') NE 0 THEN RETURN
On_Error, 2 ; Return to caller.
; Check parameters and keywords.

IF N_Elements(notifyIDs) EQ 0 THEN Message, 'Notification IDs are a required parameter.'
IF N_Elements(filename) EQ 0 THEN $
    filename=Filepath(SubDirectory=['examples','data'],'ctscan.dat')

```

```

IF N_Elements(xsize) EQ 0 THEN xsize = 256
IF N_Elements(ysize) EQ 0 THEN ysize = 256
; Create a top-level base.
tlb = Widget_Base(Column=1, Title='Enter File Information...', /Base_Align_Center)
; Make sub-bases.
subbase = Widget_Base(tlb, Column=1, Frame=1)
filebase = Widget_Base(subbase, Row=1)
; Create widgets for filename. Set text widget size appropriately.
filesize = StrLen(filename) * 1.25
fileID = FSC_InputField(filebase, Title='Filename:', Value=filename, $
    XSize=filesize, LabelSize=50, /StringValue)
browseID = Widget_Button(filebase, Value='Browse', Event_Pro='ReadImage_BrowseFiles')
xsizeID = FSC_InputField(subbase, Title='X Size:', $
    Value=xsize, /IntegerValue, LabelSize=50, Digits=4)
ysizeID = FSC_InputField(subbase, Title='Y Size:', $
    Value=ysize, /IntegerValue, LabelSize=50, Digits=4)
; Set up Tabing between fields.
fileID->SetTabNext, xsizeID->GetTextID()
xsizeID->SetTabNext, ysizeID->GetTextID()
ysizeID->SetTabNext, fileID->GetTextID()
; Make a button base with frame to hold DISMISS and APPLY buttons.
butbase = Widget_Base(tlb, Row=1)
dismissID = Widget_Button(butbase, Value='Dismiss')
applyID = Widget_Button(butbase, Value='Apply')
; Center the program on the display
screenSize = Get_Screen_Size()
geom = Widget_Info(tlb, /Geometry)
Widget_Control, tlb, $
    XOffset = (screenSize[0] / 2) - (geom.scr_xsize / 2), $
    YOffset = (screenSize[1] / 2) - (geom.scr_ysize / 2)
; Realize top-level base and all of its children.
Widget_Control, tlb, /Realize
; Create info structure to hold information needed in event handler.
info = { notifyIDs:notifyIDs, $ ; The list of widgets to notify.
        fileID:fileID, $ ; Identifier of widget holding filename.
        xsizeID:xsizeID, $ ; Identifier of widget holding xsize.
        ysizeID:ysizeID $ ; Identifier of widget holding ysize.
    }
; Store the info structure in the top-level base
Widget_Control, tlb, Set_UValue=info, /No_Copy
; Register the program, set up event loop. Make this program a
; non-blocking widget.
XManager, 'readimage', tlb, Event_Handler='ReadImage_Events', $
    /No_Block, Group_Leader=group_leader
END

```

Book Index

♦ Discovering the Possibilities ♦♦♦



Index

A

AddPath command 5
animating data 102, 104
annotating plots 52
Arg_Present command 220
arguments. *See* parameters 211
array subscripting 222
arrays 9
 changing size of 68
 creating 12
 gridding 106
 resizing 68
 square bracket subscripting 13, 223
ASCII_Template command 139
aspect ratio
 of images 71
 of windows 185
Assoc command 146
associated variables 145
 advantages of 146
 defining 146
axes
 3D 99
 adding to plot 29
 annotating 92
 box style 26
 multiple on same plot 29
 setting style of 26
 table of values for *Style* keywords 26
axes range
 setting 25
 setting exact range 25
axis
 setting tick intervals on 92
Axis command 29

B

background color
 setting 24
base widget
 top-level base 264

base widget event structure 389
base widgets
 floating 329
 modal 328
batch files
 commands in 207
BEGIN...END statement blocks 224
blocking widgets 326, 331
box
 drawing 111
 rubberband 118
box axes 26
BoxImage_Define object program code 399
BREAK statement 227
buffering 287
Butterworth frequency filter 79
button widget event structure 389, 390, 391, 392
BytScl command 62

C

capitalizing commands 3
CASE statement 227
Catch command 230
Catch error handling 230, 242
character size
 changing 247
 setting 22
CharSize keyword 22, 247
circle
 creating in IDL 57
Cleanup method of object 356, 360
Cleanup routines
 in widget programs 282
code *See* program modules 209
color 86
 dynamic displays 83
 on surface plots 32
 setting on line plots 24
 static displays 83
color aware programs 284
color bar 251
 creating 58, 70

Index

color decomposition **2**
 for image display **66**
 on or off? **86**

color displays
 depth of **83**
 dynamic **83**
 static **83**
 type of **83**

color palettes
 in HDF files **173**

color PostScript output **182**

color tables
 automatic update of **66**
 changing **89**
 creating **89**
 editing **63**
 multiple **63**
 on 24-bit displays **64**
 on 24-bit display **88**
 saving **91**
 updating with *XColors* **66**
 updating with *XLoadCT* **66**

color visual classes **83**

Color24 command **86**

Colorbar command **58, 98, 251**

colors **63, 64**
 color models in IDL **81**
 editing color tables **63**
 gray-scale only **2**
 in object programs **370**
 in PostScript files **189**
 in widget programs **301**
 Indexed Color Model **82**
 loading drawing colors **87**
 not displaying **2**
 notifying objects of change in **371**
 number in IDL session **60**
 obtaining 24-bit value **86**
 obtaining by name **87, 244**
 obtaining device-independent colors **87**
 obtaining the current color table vectors **89**
 on 24-bit display **85, 88**
 on plots **24**
 obtaining the current color vectors **285**
 protecting in widget programs **284**
 required to work with this book **2**
 RGB color model **85**
 selecting by name **302**
 selecting name of **370**

column formatted files **137**

comma separated files **141**

command continuation character **4**

commands **207**
 anatomy of **6**
 as functions **7**
 as procedures **7**
 capitalizing **3**
 collecting in journal **123**
 continuing on next line **4**

 executive **209**
 journal of **8**
 multiple **224**
 multiple on same line **224**
 on-line help for **8**
 saving **8**

comment character **3**

comments
 in IDL code **3**

common blocks
 protecting in widget programs **308**

.*Compile* command **235**

compiler options
 long integers **13**
 square bracket array subscripting **13, 223**

compiling IDL programs **235**
 from within a procedure **237**
 rules for automatic compilation **236**

compound widget event structures **393**

conditional expressions **226**

Congrid command **68**

CONTINUE statement **227**

Contour command **36**

contour plots **36**
 adding color to **41**
 algorithms for drawing **38**
 customizing **39**
 downhill direction of **41**
 filled contours **42**
 on map projections **44**
 in 3D space **42, 102**
 labeling contour levels **39**
 missing data in **96**
 positioning in window **44**
 selecting contour intervals **38**
 selecting line styles of **39**
 setting line thickness **40**

contours
 downhill direction **41**
 drawing in color **41**
 filled **42**
 labeling **38**
 selecting intervals **38**
 selecting line styles of **39**
 setting thickness of lines **40**

Convert_Coord command **58**

convex hull **107**

Convol command **76**

convolution kernels **76**

convolution of images **76**

coordinate system
 data **52**
 device **52**
 normalized **52**

coordinate systems **19**
 converting from one to another **58**

coordinates
 converting from one system to another **58**

Copy keyword **116**

CopyData command 5
Coyote's Guide to IDL Programming web page 6
 crashes
 recovering from 283
 current graphics window 109
 cursor 109
 behavior 110
 drawing a box with 111
 for annotating plots 111
 positioning in window 110
 with images 112
Cursor command 109
 in draw widgets 265
CW_Field command 329
 compound widgets
 329
D
!D.N_Colors system variable 60
 data
 encapsulation 350
 formatted 133
 gridding 106
 missing 95
 not a number 96
 range of 25
 types of 10
 unformatted 142
 data animation 102
 data coordinate system 52
 converting to device 58
 data files 4
 column format 137, 139
 copying from IDL distribution 5
 downloading 5
 installing 5
 locating 130
 reading ASCII data 140
 reading with associated variables 145
 selecting 130
 selection 130
 skipping records in 134
 template for reading 139
 unformatted 142
 decomposition on or off 86
 Delaunay triangulation 106
Device command 116, 176
 device coordinate system 52
device copy method of erasing 114, 116, 287
Dialog_Pickfile command 130
Dialog_PrinterSetup command 321
Dialog_PrinterSetup command 176, 200
 dialogs
 modal 327
 non-modal 340
 DICOM files
 reading 157
 using the IDLffrDICOM object 157
 DirectColor visual class 83
 directory
 home 4
 directory name
 selecting 131
 documentation
 on-line 8
 double buffering 287
 Draw method of objects 362
 draw widget event structure 390
 draw widgets
 creating 265
 making current window 266
 using *Cursor* command in 265
 value of 266
 window index number of 266
 drawing color
 setting 24
 droplist widget event structure 390
 dynamic color displays 83
E
 edge enhancement
 of images 77
 encapsulated PostScript output 181
 encapsulated PostScript preview 182
 encapsulation
 in objects 350
Erase command 18
 in PostScript file 180
 erasing display window 60, 114
 erasing graphics windows 18
 error condition
 generating 212
 error handing
 with *Catch* 358
 error handling 212, 229
 example of catching error 231
 generating the error condition 233
 hierarchy of handling 231
 in programs 242
 tracing the error 234
 with *Catch* 230
 with *Error_Message* 234
 with *On_Error* 229
 with *On_IOError* 229
Error_Message command 234, 242
 errors
 file I/O 229
 generating 233
 generating error messages 233
 handling with *Error_Message* 234
 program 230
 recovering from 230, 283
 reporting 232
 tracing 234
 event driven programs 260
 event handler

Index

assigning to top-level base 265
event handler module 260
event handler modules
 example for Quit button 279
 example for resizable graphics window 280
 writing 276
event handlers
 as functions 278
 assigning to the top-level base 279
 assigning to widgets 279
 naming 279
event loop 261
 creating 275
event structure
 for base widget 389
 for button widget 389, 390, 391, 392
 for compound widgets 393
 for draw widget 390
 for droplist widget 390
 for keyboard focus events 395
 for kill widget events 395
 for label widget 390
 for list widget 391
 for slider widget 391
 for table widget 391
 for text widget 392
 for timer events 395
 for tracking events 395
event structures 261
 fields in 277
 Handler field 277
 ID field 277
 Top field 277
Event_Handler keyword 265
events
 creating pseudo events 342
 sending events to widgets 344
 sending from other programs 306
 widget 261
exclusive OR method of erasing 114
executing in batch mode 207
executive commands 209
explicit file formats 140
 `_Extra` keyword 379

F

false condition in IDL 223
Fanning Software Consulting
 contacting 6
Fast Fourier transform 78
FFT command 78
file headers 137, 144
file I/O errors 229
file names
 constructing 131
 specifying in device-independent way 131
file path 131
file pointers 134

Filepath command 131
files
 help with 133
 locating 130, 131
 logical unit numbers of 131
 opening for reading 129
 opening for updating 129
 opening for writing 129
 selecting 130
 selecting names of
 used with this book 4
files *See also* data files
filled contours 42
filtering
 of images 78
filters
 building image filters 78
Findfile command 131
Floating keyword 329
floating widgets 329
Follow keyword 38
fonts
 hardware 52
 Hershey 187
 names of available hardware fonts 52
 names of available true-type fonts 52
 names of available vector fonts 52
 PostScript 187
 rotating 53
 selecting 53
 table of 53
 table of Hershey to PostScript 189
 true-type 51
FOR loops 226
Format keyword 140
format specifiers 140
formatted data
 explicitly formatted 140
 reading 133
 writing 133
Forward_Function command 223
Free_Lun command 132
frequency domain filtering 78
FSC_PSCConfig object 350
FSC_Window command 256
ftp
 downloading book files 5
functions
 calling 7
 declaring for compiler 223
 writing 221

G

Get_Lun command 132
Get_Visual_Depth keyword 83
Get_Visual_Name keyword 83
GetColor command 87, 244
GIF file output 312, 315

GIF files **147**
 color **147**
 creating **150**
 license required **150**
 writing **150, 151, 154**
Go executive command **209**
GOTO statement **229**
 graphic margin **44**
 graphic position **44**
 graphic region **44**
 graphics
 buffering display of **287**
 device copy method of erasing **114**
 in resizeable windows **256**
 object **19**
 positioning in window **246**
 raster **19**
 graphics device
 an X Windows display (X) **176**
 CGM **176**
 default setup **176**
 hardcopy output **175**
 HPGL plotter **176**
 PCL printer **176**
 PostScript **176**
 graphics devices
 CGM **176**
 HP **176**
 MAC **176**
 PCL **176**
 PRINTER **176**
 PS **176**
 WIN **176**
 X **176**
 Z **176**
 graphics display **287**
 graphics function **114**
 graphics window
 in widget programs **265**
 pixmap **116**
 graphics window. *See* window
 grid lines on plots **27**
 gridding
 Delaunay triangulation method **106**
 spherical **108**
 gridding data **106**
 group leaders
 in widget programs **310**

specifying sizes of **178**
 hardware fonts **52**
 names of **52**
 HDF data **161**
 description of data objects **164**
 self-describing format **162**
 tags **163**
 types of data objects **163**
 HDF data files
 adding color palettes to **173**
 closing **166**
 closing SDS files **169**
 creating a new SDS **169**
 creating attributes **170**
 creating SDS files **169**
 defining attributes for **167**
 dimension scales **167**
 number of tags in **166**
 opening **165**
 opening SDS files **169**
 predefined attributes of **167**
 scientific data sets in **166**
 selecting SDS files **169**
 table of routines for **168**
HDFRead program code **406**
HDFWrite program code **407**
 headers **137, 144**
 heap variables
 pointers **268**
Heap_GC **271**
 help
 contacting the author **6**
Coyote's Guide to IDL Programming **6**
 heap variables **351**
 on-line **8**
 with files **133**
 with objects **351**
 Hershey fonts **53, 187**
 hierarchical data format *See* HDF data
 high-pass filter **79**
Hist_Equal command **74**
Histogram command **248**
 histogram equalization **74**
 histogram plots
 displaying **248**
HistoImage program **240**
 hourglass cursor
 setting **321**

H

hardcopy output **175**
 closing the file **177**
 closing the printer document **177**
 controlling the device **176**
 landscape mode **178**
 portrait mode **178**
 selecting a file name **177**
 selecting graphics device **175**

I

IDL
 required version **2**
 IDL code
 supplied with the book **261**
 IDL commands. *See* commands
 IDL home directory **4**
 IDL programs
 source code of **399**

Index

- IDL source code **399**
 - IF statements **225**
 - IF...THEN...ELSE statements **225**
 - image data **59**
 - scaling **62**
 - image processing **74**
 - filtering **78**
 - histogram equalization **74**
 - in objects **372**
 - smoothing **75**
 - image registration **123**
 - images
 - (0,0) point in **64**
 - 24-bit images **64**
 - 24-bit on 8-bit display **65**
 - band interleaved **64**
 - changing size of **68**
 - colors in PostScript **183**
 - convolution of **76**
 - display order **68**
 - displayed in PostScript files **193**
 - displaying **60**
 - displaying 8-bit on 24-bit display **66**
 - edge enhancement **77**
 - filters for **78**
 - in 24-bit environment **64**
 - pixel interleaved **64**
 - positioning in window **69**
 - PostScript **183**
 - reading from display **72**
 - removing noise from **77**
 - row interleaved **64**
 - scaling **62**
 - sizing in PostScript files **69, 194**
 - true-color **64, 183**
 - upside down **68**
 - using cursor with **112**
 - warping **123**
 - Indexed color model **81**
 - info* structure
 - creating **267**
 - inheritance
 - in objects **374**
 - inheriting object methods **375**
 - inheriting structures **375**
 - Init method of object **356**
 - initialization
 - of objects **361**
 - integers
 - forcing four-byte integers **13**
- J**
- Journal* command **8**
 - journal file **123**
 - journal of IDL commands **8**
 - JPEG file output **312, 315**
 - JPEG files **147**
 - color **154**
- creating **154**
 - reading **155**
 - writing **155**
- K**
- Keep_Aspect_Ratio* keyword **71**
 - keyboard focus event structure **395**
 - keyboard focus events
 - in widget programs **285**
 - keyword inheritance **216, 219**
 - keyword parameters **7**
 - keyword parameters. *See also* parameters
 - Keyword_Set* command **215**
 - keywords
 - as optional parameters **214**
 - checking for **215, 242, 243**
 - defined? **214**
 - defining
 - inheritance **216**
 - inherited **219**
 - passing to other commands **216**
 - present? **220**
 - used? **214**
 - with binary properties **215**
 - kill widget event structure **395**
- L**
- label widget event structure **390**
 - labels
 - on contour plots **39**
 - lifecycle methods of objects **361**
 - line plots **20**
 - annotating **52**
 - axes style **26**
 - drawing lines on **56**
 - drawing symbols on **56**
 - establishing another axis on **29**
 - filling with color **57**
 - grids on **27**
 - limiting data range on **25**
 - line styles for **22**
 - margin around **46**
 - missing data in **96**
 - multiple data sets on **28**
 - multiple in window **47**
 - plotting symbols **23**
 - position of **46**
 - region of **47**
 - using color with **24**
 - line plots *See also* plots
 - line styles
 - selecting **22**
 - table of **22**
 - line symbols **23**
 - line thickness
 - setting **22**
 - list widget event structure **390**
 - LoadData* command **20**

logical unit numbers 131
 assigning 131
 freeing 132
 obtaining with *Get_Lun* keyword 131
 reusing 132

loops
 breaking out of 227
 going to next iteration 227
 low-pass filter 79

M

MAC device 176
 main-level program 208
Margin keyword 46
Max command 60
 maximum value 60
Median command 75
 memory
 clean up 271
 leaking 271
 menu buttons
 creating 265
 menubar 265
 menus
 pull-down 265
Message command 212, 233
 method overriding 374
 methods
 attaching methods to objects 375
 in superclass objects 375
 lifecycle methods 361
 of objects 350
 overriding 374
 restrictions to attaching 378
 to get properties of objects 364
 to set properties of objects 364
Min command 60
 minimum value 60
 missing data 95
Modal keyword 327
 modal widget programs 327
 modal widgets
 storing information in 330
!Mouse system variable 111
 mouse. *See* cursor
 MPEG movies 105
 multiple axes on plot 29
 multiple color tables 63, 64
 multiple plots in window 47

N

N_Elements command 215
NaN value 96
No_Block keyword 326
NoErase keyword 47
 noise
 removing from images 77
 non-modal dialogs

communicating with 342
 normalized coordinate system 52
 Not a Number value 96
NotifyObj keyword 371

O

Obj_Destroy command 353
Obj_New command 350
 object
 Cleanup method 356
 Init method 356
 lifecycle methods 356
 self variable 360
 object class 353
 object encapsulation 350
 object graphics 19
 object inheritance 374, 375
 object methods 350
 object oriented programming 349
 object polymorphism 380
 objects
 attaching superclass methods to 375
 Cleanup method 360
 common problems creating 361
 creating in IDL 349, 350
 defining object class 353
 destroying in IDL 353
 Draw method 362
 freeing memory in 357
 getting properties of 364
 image processing in 372
 inheritance in 374
 INIT method 357
 initialization parameters 361
 initializing 357
 lifecycle methods of 361
 notifying when colors are loaded 371
 polymorphism in 380
 setting properties of 364
 specific instance of 360
 subclass 374
 superclass 374
 working with colors in 370
On_Error command 230
On_IOError command 229
 opening files 129
OpenR command 129
OpenU command 129
OpenW command 129
OPlot command 25, 28
!Order system variable 64, 68
 output *See* hardcopy output
 overplotting data on line plots 28
 overriding object methods 374

P

palette objects 163
 parameters

Index

checking for 242
defining optional 212
defining required 212
keyword 7, 213
output 217
passing by reference 217
passing by value 217
positional 6, 211
parent widget 264
perimeter (of set of points) 107
PickColorName command 302
PickColorName command 370
Pickfile command *See Dialog_Pickfile* command
pixmaps 116, 119
 for animation 104
Plot command 20
plot margin 46
plot position 46
plot range
 setting 25
plot region 47
plot symbols
 creating your own 24
 table of 24
plots
 annotating 52, 56
 arranging multiple 50
 axes style 26
 changing line style 22
 color with 24, 57
 colors in PostScript files 189
 colors on 24-bit display 85
 combining 100
 contour 36
 customizing 22
 formatting tick labels 92, 94
 grids on 27
 in color 24
 limiting data range on 25
 line 20
 line thickness 22
 missing data in 96
 multiple in window 47
 no erasing first 47
 positioning in window 44
 scatter plot 98
 setting axis properties of 26
 setting graphics function 114
 setting range of 25
 setting symbol size 24
 symbols on 23
 tick marks (controlling) 27
 titles on 21
 titles on multiple plots 48
 using the cursor with 109
 with grid 27
PlotS command 56
plotting symbols
 creating your own 24
 table of 23
!P.Multi system variable 47
Point_Lun command 134
pointer variables 268
pointers
 cleaning up in widget programs 282
 creating 269
 de-referencing 269
 freeing 269
 in IDL 268
 in widget programs 271, 330
 leaking memory from 271
 memory cleanup 271
 null pointer 269
 releasing memory from 269
 to undefined variable 270
 valid pointers 269
Polyfill command 57, 125, 190
polymorphism in objects 380
PolyShade command 126
position
 of graphic in window 44, 45
Position keyword 45
PostScript device
 configuring interactively 198
PostScript file output 312, 317
PostScript files
 printing on Macintosh 181
 printing on Windows 181
PostScript fonts 187
PostScript graphics device 176
PostScript output
 color 182
 configuring 314
 encapsulated 181
 landscape mode offsets 198
 offsets 198
 preview mode 182
 printing from within IDL 180
 resizing images in 69
!P.Position system variable 45, 46
!P.Region system variable 47
printer
 configuring 321
Printer device 176
 color bug in 252
 color loading bug 204
 configuring 200
 loading colors in 204
 positioning images on 203
 positioning output on 202
PrintF command 133
printing
 directly from programs 320
 set up for 321
procedure definition statement 209
procedures
 calling 7
 writing 209

program control statements 223
 program crashes
 recovering from 283
 program errors 230
 recovering from 230
 program files 4
 downloading 5
 program modules
 compiling 235
 functions 221
 main-level program 208
 procedures 209
 programs
 crashed 230
 event driven 260
 recovering from program errors 230
 saving as compiled 237
 source code of 399
 supplied with the book 261
 widget 260
PSCfg program 314
PSCfg program 198
 pseudo events 342, 344
 PseudoColor visual class 83
PSym keyword 23
!P.T system variable 97
Ptr_Free command 269
Ptr_New command 269
Ptr_Valid command 269
 pull-down menu 265
 creating 294, 301, 312, 320

R

random values 98
RandomU command 98
 range
 of axes 25
 raster graphics 19
 raster image objects 163
Read command 134
Read_ASCII command 140
ReadF command 134
 reading data
 from a string 141
 reading files 129
 reading formatted data
 examples of 136
 rules for 134
ReadS command 141
ReadU command 142
 realizing widgets 266
Rebin command 68
_Ref_Extra keyword 379
 registering widget programs 275
 REPEAT...UNTIL statements 227
 Research Systems
 contacting 2
.Reset_Session executive command 354

resetting the IDL session 354
 resizable graphics windows 280
Resolve_All command 237
Resolve_Routine command 237
Restore command 91, 237
 restoring compiled programs 237
RetAll command 283
RetAll command 230
 RGB 85
 RGB color model 81
.RNew command 235
Roberts command 77
 rotating surface plots 31
 rubberband box 118
.Run command 235

S

Save command 91, 237
 saving compiled programs 237
 saving IDL commands 8
 scalars 9
Scale3 command 99
 scaling data 62
 scatter plot 98
 scientific data objects 163
 scientific data sets *See* HDF data files
 scope of program variables 210
 screen dumps 72
 scripts *See* program modules
 scrolling
 graphics windows 119
 SDS files *See* HDF data files
 self object 360
Send_Event keyword 306
Send_Event keyword 344
 session
 resetting the IDL session 354
Set_Graphics_Function keyword 114
Set_Plot command 175
Set_Shading command 34, 35
Shade_Surf command 34
Shade_Volume command 126
 shaded surfaces 34
 changing parameters of 34
 shading
 changing parameters for 35
 slider widget event structure 391
Smooth command 75
 smoothing images 75
Sobel command 77
 software fonts 53
 source code 399
 spherical gridding 108
 static color displays 83
Str_Size command 247
 structure inheritance 375
 structures 9
 automatic definition of 355

Index

- definition statement **354**
- determining names of **332**
- in IDL **353**
- named **353**
- review of **353**
- subclass object **374**
- subscripted variables
 - reading into **138**
- subscripts **55**
- superclass object **374**
- superclass object methods **375**
- superscripts **55**
- Surface* command **29**
- surface plots **31**
 - adding color to **32**
 - adding skirt to **34**
 - changing appearance of **34**
 - changing shading parameters **35**
 - customizing **31**
 - draping with data **36**
 - draping with other data sets **32**
 - missing data in **96**
 - rotating **31**
 - shaded **34**
 - skirts on **34**
 - wire mesh **29**
- SWITCH statement **228**
- symbols
 - creating your own **24**
 - for plotting **23**
 - table of plot symbols **24**
- SymSize* keyword **24**

- T**
- T3D* command **97**
- tabling in widget programs **336**
- table widget event structures **391**
- Tag_Names* command **287, 332**
- test
 - superscripts **55**
- text
 - adding to plots **54**
 - aligning **54**
 - erasing **54**
 - orienting **54**
 - positioning on line **55**
 - rotating **55**
 - subscripts **55**
- text size
 - changing **247**
- text widget event structures **392**
- 3D coordinate system **97**
- 3D coordinates
 - axes for **99**
- 3D graphics **97, 120**
- tick labels **92**
 - formatting **94**
- tick marks **92**
- inward facing **27**
- outward facing **27**
- size of **27**
- TIFF file output **312, 316**
- TIFF files
 - creating **156**
- timer event structure **395**
- titles
 - on line plots **21**
- top-level base widget **264**
- tracking event structure **395**
- transparency **125**
- Triangulate* command **106**
- TriGrid* command **106**
- true condition in IDL **223**
- TrueColor visual class **83**
- true-type fonts **51**
 - names of **52**
- TV* command **59, 60**
- TVImage* command **71, 100**
- TVLCT* command **285**
- TVLCT* command **89**
- TVRD* command **72**
- TVScl* command **59, 60**

- U**
- undo capability
 - in widget programs **298**
- Undo object method **373**
- unformatted data
 - example of reading **144**
 - example of writing **143**
 - reading **142**
 - writing **142**
- unformatted data files **142**
 - problems with **145**
 - reading with associated variables **145**
- unsharp masking **76**
- user values
 - to store information **271**

- V**
- variables
 - attributes of **9**
 - basic types of in IDL **10**
 - changing attributes dynamically **12**
 - creating **9**
 - disappearing **230**
 - in pointers **268**
 - passing by reference **217**
 - passing by value **217**
 - scope of **210**
 - valid names for **9**
- Vdata* objects **163**
- vector fonts **53, 187**
 - names of **52**
- vectors **9**
 - creating **12**

version of IDL required 2
 Vgroup objects 163
 volume rendering 126

W

warping images 123
Where command 137
 WHILE loops 226
 widget definition module 259, 260
 purposes of 261
 writing 261
 widget event handler module 259
 widget event structure 261
 widget events 261
 sending from other programs 306
 widget family tree 263
 widget hierarchy 263
 top-level base of 264
 widget programs
 "memory" in 297
 blocking 326
 clean up routine 282
 cleaning up 311
 collecting information in 325
 controlling colors in 301
 dialogs in 325
 event handler module 260
 event handlers in 260
 file output from 312
 flow of information through 259
 group leaders in 310
 implementing an Undo function 298
 keyboard focus events 285
 memory management 282
 modal 327
 non-modal dialogs 340
 realizing 266
 recovering from program errors 283
 registering with window manager 275
 sending events from 306
 storing information in 267, 271
 structure of 259
 tabbing in 336
 using pointers in 271
 widget definition module 260, 261
 writing 259
 widget user values
 to store information 271
Widget_Control command 266
 widgets
 blocking 326, 328, 331
 creating 263
 creating pseudo events 344
 currently registered 308
 dialog form widgets 325
 floating 329
 hierarchies of 263
 modal 326, 327, 328

modal base 328
 non-blocking 326
 non-modal dialogs 340
 non-modal forms 340
 notifying of program actions 342
 window scrolling 119
 windows
 erasing 60
 in widget programs 265
 pixmap 116
 positioning images in 69
 resizeable 256, 280
 smart graphics command 256
 when not to open them 255
 wire surface plots 31
WriteU command 142
 writing files 129
 writing formatted data
 examples of 136

X

XColors command 66, 306
XColors event structure 394
XInterAnimate command 102
XLoadCT command 89
 XOR mode 114
XPalette command 89
XRegistered command 308
XWindow command 200
XYOutS command 52, 95, 247

Z

Z device 176
 Z-graphics buffer 120
 configuring 121
 transparency effects 125