# IDL

# Application Programming

0509IDL71BLD

# Contents

# Chapter 6
# Library Authoring ............................................................. 105

# Chapter 7
# Program Control ............................................................... 113

# Chapter 8
# Debugging and Error-Handling ....................................... 141

## Part II: Components of the IDL Language

## Chapter 13
## Working with Data in IDL .................................................... 247

## Chapter 14
## Strings ................................................................................. 275

## Chapter 15
## Arrays ................................................................................. 301

## Chapter 16
## Structures ....................................................................... 337

## Chapter 17
## Pointers ........................................................................... 359

# Part III: Creating Applications in IDL

# Part I: Application Programming

# Chapter 1
# Overview of IDL Applications

This chapter includes information about the following topics:

# What is an IDL Application?

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

## Can I Distribute My Application?

You can freely distribute IDL source code for your IDL applications to colleagues and others who use IDL. (If you intend to distribute your applications, it is a good idea to avoid any code that depends on the qualities of a specific platform. See "!VERSION" (*IDL Reference Guide*) and "Tips on Creating Widget Applications" (Chapter 3, *User Interface Programming*) for some hints on writing platform-independent code.) Of course, IDL applications can only be run from within the IDL environment, so anyone who wishes to run your IDL application must have access to an IDL license.

If you would like to distribute your IDL application to people who do not have access to an IDL license, you have several options. Many IDL applications will run in the freely-available IDL Virtual Machine. If your application uses features not available in the virtual machine, you may wish to consider a *runtime IDL* licensing agreement. Runtime IDL licenses allow you to distribute a special version of IDL along with your application. See Chapter 23, "Distributing Runtime Mode Applications" for a complete discussion of the different ways you can distribute an application written in IDL.

# About Building Applications in IDL

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

Advantages of IDL include:

- IDL is a complete, structured language that can be used both interactively and to create sophisticated functions, procedures, and applications.

- Operators and functions work on entire arrays (without using loops), simplifying interactive analysis and reducing programming time.

- Immediate compilation and execution of IDL commands provides instant feedback and "hands-on" interaction.

- Rapid 2D plotting, multi-dimensional plotting, volume visualization, image display, and animation allow you to observe the results of your computations immediately.

- Many numerical and statistical analysis routines—including Numerical Recipes routines—are provided for analysis and simulation of data.

- IDL's flexible input/output facilities allow you to read any type of custom data format. Support is also provided for common image standards (including BMP, JPEG, and XWD) and scientific data formats (CDF, HDF, and NetCDF).

- IDL widgets can be used to quickly create multi-platform graphical user interfaces to your IDL programs.

- IDL programs run the same across all supported platforms (Microsoft Windows and a wide variety of Unix systems) with little or no modification. This application portability allows you to easily support a variety of computers.

- Existing FORTRAN and C routines can be dynamically-linked into IDL to add specialized functionality. Alternatively, C and FORTRAN programs can call IDL routines as a subroutine library or display "engine".

# Chapter 2
# Creating and Running Programs in IDL

The following topics are covered in this chapter:

# Overview of IDL Program Types

In addition to being a useful interactive data analysis tool, IDL is a powerful programming language. Many of IDL's programming language features and constructs can be used either interactively at the IDL command line or as part of a larger program — which can itself be invoked at the IDL command line or by other programs. A program may or may not be compiled before execution. The type of programs you use in IDL will depend upon your tasks.

| Program Type | Description |
|---|---|
| iTools State File (`.isv`) | Restore or share an iTools session — you can save the current state of an iTool as an *iTools State* (*.isv) file. Whenever you close an iTool window, you are prompted to save the current state as an *.isv file so that you can return to the current state of the data later when you open the *.isv file. Other IDL users running the same version or a newer version of IDL can open *.isv files. The iTool State file includes the data visualized at the time it was created. There is no need to provide a separate data file to support the visualization. See the *iTool User's Guide* for details. |
| $MAIN$ Program | Repeat a series of command line statements or interactively change variable values in a program file. These short programs or procedures are called $MAIN$ (main-level) programs. They are not explicitly named, and cannot be called from other programs. See "Creating $MAIN$ Programs" on page 22 for details. |
| Named Program File (`.pro`) | Create programs and applications — you can create programs for data analysis or visualization using one or more *named program files* (*.pro). Program files are created in the IDL Editor or a text editor of your choice. See "About Named Programs" on page 25. |

*Table 2-1: IDL Program Types*

| Program Type | Description |
|---|---|
| Batch File | Automate processing tasks — you can automate routine or lengthy processing tasks using a *batch file*, which contains one or more IDL statements or commands. Each line of the file is read and executed before proceeding to the next line. See Chapter 3, "Executing Batch Jobs in IDL" for additional information on batch files. |
| SAVE File (.sav) | Share programs and distribute applications — you can create a SAVE file containing data or named program files in a .sav file to share with other users who may or may not have a full IDL installation. See Chapter 4, "Creating SAVE Files of Programs and Data" for details. |

*Table 2-1: IDL Program Types (Continued)*

# Creating $MAIN$ Programs

A $MAIN$ (main-level) program can be created in two ways: at the command line and in a text editor. You typically create a $MAIN$ program at the IDL command line when you have a few commands you want to run without creating a separate file to contain them. Creating a $MAIN$ program in a text file allows you to combine the functionality of named procedures and functions with the ability to have command line access to variable data that is defined in the $MAIN$ scope.

$MAIN$ programs are not explicitly named; they consist of a series of statements that are not preceded by a procedure (PRO) or function (FUNCTION) heading. They do, however, require an END statement. Since there is no heading, the program cannot be called from other routines and cannot be passed arguments. When IDL encounters a main program either as the result of a .RUN executive command, or in a text file, it compiles it into the special program named $MAIN$ and immediately executes it. Afterwards, it can be executed again using the .GO executive command.

## Creating a $MAIN$ Program at the Command Line

To create a $MAIN$ level program at the command line, start IDL and complete the following steps:

1. **Initialize a variable.** At the IDL command line, enter the following:

   ```
   A = 2
   ```

2. **Designate a command line $MAIN$ program.** Enter .RUN at the IDL command line:

   ```
   .RUN
   ```

   The command line prompt changes from IDL> to -.

3. **Enter the program statements.** Create a $MAIN$ level program consisting of the following statements:

   ```
   A = A * 2
   PRINT, A
   END
   ```

   The $MAIN$ program is immediately compiled and executed when you enter the END statement. IDL prints 4.

4. **Re-execute the $MAIN program.** Enter .GO at the IDL command line:

   ```
   .GO
   ```

   The $MAIN$ program is executed again, and now IDL prints 8.

# Creating a $MAIN$ Program in a Text File

When you create a $MAIN$ program in a named text file, you can execute the program and have command line access to variables. This is an easy way to run and test various variable values without having to modify the code and rerun the entire program, or set breakpoints. The following example allows you to create, save, run and test a $MAIN$ program.

1. **Create the $MAIN$ program file.** Enter the following into the IDL Editor. This example consists of a function that modifies the image data, and a $MAIN$ program. The $MAIN program displays the original image, solicits a threshold value, passes the value to the function, and displays the new image data:

```
FUNCTION stretchImage, img, value

; Stretch image by input amount.
image = img > value
RETURN, image

End

; --- Begin $MAIN$ program.--------------------
; Display the image, solicit threshold value and
; display new results.

; Set up display.
DEVICE, DECOMPOSED = 0, RETAIN = 2
LOADCT, 0

; Access image data and display.
img = READ_PNG(FILEPATH('mineral.png', $
   SUBDIRECTORY = ['examples', 'data']))
dims = SIZE(img, /DIMENSIONS)
WINDOW, 0, XSIZE = dims[0], YSIZE = dims[1]
TVSCL, img

; Ask for a threshold value and stretch image.
READ, threshold, PROMPT='Enter Numerical Value: '
newImg = stretchImage(threshold, img)

; Display the results.
TVSCL, newImg

END
```

2.  **Save the $MAIN$ program.** Save the file as interactivestretch.pro. It is important to note that a $MAIN$ program should *not* have the same name as any internal procedures or functions.

3.  **Run the $MAIN program.** Type the following at the command line to run the program:

    ```
    .RUN interactiveStretch.pro
    ```

    This compiles internal functions and procedures, and executes the $MAIN program. The command line prompt changes from IDL> to -.

4.  **Enter a threshold value.** Enter 67 (or any value between 0–255) at the command line and press **Enter**. This scales the image so that the remaining pixel values are stretched across all possible intensities (0 to 255).

5.  **Test another threshold value.** Enter .GO at the IDL command line:

    ```
    .GO
    ```

    Enter a different value and press enter to see the results. These two final steps can be repeated as many times as you like.

# About Named Programs

Longer routines and programs, consisting of more than a few lines, are typically given their own explicit names, allowing them to be called from other programs as well as executed at the IDL command line. Named programs are stored in disk files created using a text editor. The IDL Workbench includes a built-in text editor, but any text editor can be used to create named IDL programs. Files containing IDL programs, procedures, and functions are assumed to have the filename extension `.pro`.

**Note** ────────────────────────────────────────────────────────────
Although any text editor can be used to create an IDL program file, the IDL Editor contains features that simplify the process of writing IDL code. See "Command Line Tips and Tricks" on page 35 for details on using the IDL Editor.

─────────────────────────────────────────────────────────────────────

Most IDL applications consist of one or more IDL procedures, functions, object definitions, and object method routines:

- **Procedures** — a procedure is a self-contained sequence of IDL statements with a unique name that performs a well-defined task. Procedures are defined with the procedure definition statement, PRO.

- **Functions** — a function is a self-contained sequence of IDL statements that performs a well-defined task and returns a value to the calling program unit when it is executed. Functions are defined with the function definition statement FUNCTION.

- **Object definitions** — an object definition describes an IDL *object*, which can encapsulate both *instance data* and *method routines*. For additional information on IDL's object-oriented programming features, see Chapter 1, "The Basics of Using Objects in IDL" (*Object Programming*).

- **Object methods** — these routines are procedures and functions that act on object instance data. See "Acting on Objects Using Methods" (Chapter 1, *Object Programming*) for additional information.

See the following section for a simple procedure that calls a function. See Chapter 5, "Creating Procedures and Functions" for details on creating and calling procedures and functions, defining argument and keyword parameters, and using keyword inheritance.

**Note** ────────────────────────────────────────────────────────────
See Chapter 6, "Library Authoring" for information on procedure naming.

─────────────────────────────────────────────────────────────────────

# Creating a Simple Program

In this section, we'll create a simple "Hello World" program consisting of two .pro files. Start the IDL Workbench and complete the steps described below.

**Note** ────────────────────────────────────────────────────────
For information on using the IDL Editor, see
────────────────────────────────────────────────────────────────

1. **Open a new IDL Source File.** Start the IDL Editor by selecting **File → IDL Source File** or clicking the **New IDL Source File** button on the toolbar.

2. **Create a procedure.** Type the following in the IDL Editor:

   ```
   PRO hello_main
      name = ''
      READ, name, PROMPT='Enter Name: '
      str = HELLO_WHO(name)
      PRINT, str
   END
   ```

3. **Save the procedure.** To save the file, select **File → Save** or click the **Save** button on the toolbar. Save the file with the name hello_main.pro in the main IDL directory (which the Save As dialog should already show).

4. **Create a function.** Open a new IDL source file by selecting **File → IDL Source File** or clicking the **New IDL Source File** button on the toolbar. Enter the following code:

   ```
   FUNCTION hello_who, who
      RETURN, 'Hello ' + who
   END
   ```

5. **Save the function.** Save the file as hello_who.pro in the main IDL directory. This simple program, consisting of a user-defined procedure, calls a user-defined function.

6. **Compile the programs.** Compile hello_main.pro and hello_who.pro programs by selecting **Project → Build All**.

   **Note** ──────────────────────────────────────────────────
   You can also type .COMPILE hello_who.pro, hello_main.pro at the IDL command prompt to compile the files. With functions, the compilation order does matter. See "Compiling Your Program" on page 30 for details.
   ──────────────────────────────────────────────────────────

7. **Run the program.** Select **Run → Run hello_main**.

8.  **Enter a name.** Type your name at the IDL command line, which now reads "Enter Name" and press the **Enter** key. This passes the text to the function `hello_who`. The "Hello name" string is returned to the procedure and printed in the Console View.

# Running Named Programs

IDL program files, identified with a `.pro` extension, can be compiled and executed using the following methods:

- Running Programs Using the IDL Workbench Interface
- Running Programs From the IDL Command Line
- Running Programs Using Executive Commands

## Running Programs Using the IDL Workbench Interface

To run an IDL program using the IDL Workbench interface, do the following:

1. Open the file in the IDL Editor. For example, select:
   **File → Open File**

   and select `examples/demo/demosrc/d_uscensus.pro` from your IDL installation directory.

2. Compile the file by selecting **Run → Compile** *filename*

   where *filename* is the name of the file opened in the IDL Editor (`d_uscensus.pro`, in this example).

3. Execute the file by selecting **Run → Run** *filename*

   where *filename* is the name of the file opened in the IDL Editor (`d_uscensus.pro`, in this example).

## Running Programs From the IDL Command Line

When a file is specified by typing only the filename at the IDL prompt, IDL searches the current directory for *filename.pro* (where *filename* is the file specified) and then for *filename.sav*. If no file is found in the current directory, IDL searches in the same way in each directory specified by !PATH. If a file is found, IDL automatically compiles the contents and executes any functions or procedures that have the same name as the file specified (excluding the extension). See "Automatic Compilation" on page 30 for additional details.

Using the previous example, run the US Census Data demo by entering the following at the command line:

```
d_uscensus
```

# Running Programs Using Executive Commands

When a file is specified using either the .RUN, .RNEW, .COMPILE, or @ command followed by the filename, IDL searches the current directory for `filename.pro` (where *filename* is the file specified) and then for `filename.sav`. If no file is found in the current directory, IDL searches in the same way in each directory specified by !PATH. If a file is found, IDL compiles or runs the file as specified by the executive command used. Executive commands can be entered only at the IDL command prompt, and are often used when executing $MAIN$ program files. See "About Executive Commands" on page 38 for more information.

**Note** —————————————————————————————————————

If you are compiling files that do not exist in your path, make sure to compile functions before procedures. This keeps IDL from misinterpreting a function call as a subscripted variable or array definition. See "Compiling Your Program" on page 30 for details.

—————————————————————————————————————

**Warning** ————————————————————————————————————

If the current directory contains a subdirectory with the same name as *filename*, IDL will consider the file to have been found and stop searching. To avoid this problem, specify the extension (`.pro` or `.sav`, usually) when entering the run, compile, or batch file executive command.

—————————————————————————————————————

The details of how !PATH is initialized and used differ between the various operating systems, although the overall concept is the same. See "!PATH" (Appendix D, *IDL Reference Guide*) for more information.

# Compiling Your Program

Before a procedure or function can be executed, it must be *compiled*. When a system routine (a function or procedure built into IDL, such as iPLOT) is called, either from the command line or from another procedure, IDL already knows about this routine and compiles it automatically. When a user-defined function or procedure is called, IDL must find the routine and then compile it. Compilation can be either *automatic* or *manual*, as described below.

**Warning** ────────────────────────────────────────────────

User-written functions must be defined before they are referenced, unless they:
   1) Exist in the IDL !PATH.
   2) Exist in a `.pro` file with the same name as the function.
   3) Are reserved using the FORWARD_FUNCTION statement.
Defining the function is necessary to distinguish between function calls and subscripted variable references. See "About Calling and Compiling Functions" on page 80 for details.

────────────────────────────────────────────────────────────

## Automatic Compilation

When you enter the name of an uncompiled user-defined routine at the command line or call the routine from another routine, IDL searches the current directory for *filename*.pro, then *filename*.sav, where *filename* is the name of the specified routine. If no file is found in the current directory, IDL searches each directory specified by !PATH. (For more on the IDL path, see "!PATH" (*IDL Reference Guide*).)

If no file matching the routine name is found, IDL issues an error:

```
% Attempt to call undefined procedure/function: 'routine'
```

where *routine* is the name of the routine you specified.

If a file is found, IDL automatically compiles the contents of the file *up to the routine with the same name of the file* (excluding the suffix), and then executes the routine. If the file does not contain the definition of a routine with the same name as the file, IDL issues the same error as when the no file with the correct name is found.

For example, suppose a file named `proc1.pro` contains the following procedure definitions:

```
PRO proc1
    PRINT, 'This is proc1'
END
```

```
PRO proc2
   PRINT, 'This is proc2'
END

PRO proc3
   PRINT, 'This is proc3'
END
```

If you enter `proc1` at the IDL command line, only the `proc1` procedure will be compiled and executed. If you enter `proc2` or `proc3` at the command line, you will get an error informing you that you attempted to call an undefined procedure.

In general, the name of the IDL program file should be the same as the name of the last routine within the file. This last routine is usually the main routine, which calls all the other routines within the IDL program file (or, in the case of object classes, the class definition). Using this convention for your IDL program files ensures that all the related routines within the file are compiled before being called by the last main routine.

Program files within the IDL distribution use this formatting style. For example, open the program file for the XLOADCT procedure, `xloadct.pro`, in the IDL Editor. This file is in the `lib/utilities` subdirectory of the IDL distribution. This file contains several routines. The main routine (XLOADCT) is at the bottom of the file. When this file is compiled, the IDL Console notes all the routines within this file that are compiled:

```
IDL> .COMPILE XLOADCT
% Compiled module: XLCT_PSAVE.
% Compiled module: XLCT_ALERT_CALLER.
% Compiled module: XLCT_SHOW.
% Compiled module: XLCT_DRAW_CPS.
% Compiled module: XLCT_TRANSFER.
% Compiled module: XLOADCT_EVENT.
% Compiled module: XLOADCT.
```

Note that the main XLOADCT procedure is compiled last.

**Tip**
When editing a program file containing multiple functions and/or procedures in the IDL Editor, you can easily move to the desired function or procedure in the Outline view by selecting the Outline tab next to the Project Explorer tab. Select the function or procedure name from the list, and the Editor highlights and displays it.

# Manual Compilation

There are several ways to manually compile a procedure or function.

- Use the .COMPILE executive command at the IDL command line:

    ```
    .COMPILE myFile
    ```

    where *myFile* is the name of a .pro file located either in IDL's current working directory or in one of the directories specified by !PATH. All the routines included in the specified file will be compiled, but none will be executed automatically. If you are using the IDL Workbench, the .pro file will also be opened in the IDL Editor.

- If the file is open in the IDL Editor, select **Run → Compile** or click the **Compile** button on the toolbar. All routines within the file will be compiled, but none will be executed automatically.

- Use the .RUN or .RNEW executive command at the IDL command line:

    ```
    .RUN myFile
    ```

    where *myFile* is the name of a .pro file located either in IDL's current working directory or in one of the directories specified by !PATH. All the routines included in the specified file will be compiled, and any $MAIN$ level programs will be executed automatically. If you are using the IDL Workbench, the .pro file will also be opened in the IDL Editor.

- Use the .RUN, .RNEW, or .COMPILE executive command with no filename argument to interactively create and compile a $MAIN$ level program. The Command line prompt changes from IDL > prompt toe - so that you can start entering the $MAIN$ level program. See "Creating $MAIN$ Programs" on page 22 for additional details.

**Note** ───────────────────────────────────────────

Only .pro files can be compiled using the manual compilation mechanisms. Attempting to compile a SAVE (.sav) file using one of these mechanisms will result in an error.

───────────────────────────────────────────────────

The "Hello World" example shown in "Compiling Your Program" on page 30 has a user-defined procedure that contains a call to a user-defined function. If you enter the name of the user-defined procedure, hello_main, at the command line, IDL will compile and execute the hello_main procedure. After you provide the requested input, a call to the hello_who function is made. IDL searches for hello_who.pro, and compiles and executes the function.

# Compilation Errors

If an error occurs during compilation, the error is reported in the IDL Workbench Console view. For example, because the END statement is commented out, the following user-defined procedure will result in a compilation error:

```
PRO procedure_without_END
    PRINT, 'Hello World'
;END
```

When trying to compile this procedure (after saving it into a file named `procedure_without_END.pro`), you will receive an error similar to the following the IDL Console view:

```
IDL> .COMPILE procedure_without_END

% End of file encountered before end of program.
  At: C:\ITT\workspace\Default\procedure_without_end.pro, Line 4
% 1 Compilation error(s) in module PROCEDURE_WITHOUT_END.
```

**Note** ─────────────────────────────────────────────────

The IDL Editor displays a red dot to the left of each line that contains an error.

─────────────────────────────────────────────────

# Setting Compilation Options

The COMPILE_OPT statement allows you to give the IDL compiler information that changes some of the default rules for compiling the function or procedure within which the COMPILE_OPT statement appears. The syntax of COMPILE_OPT is as follows:

COMPILE_OPT $opt_1$ [,$opt_2$, ..., $opt_n$]

where $opt_n$ is any of the available options documented in "COMPILE_OPT" (*IDL Reference Guide*). These options allow you to change default values of true and false, hide routines from HELP, and reserve the use of parentheses for functions. See COMPILE_OPT for complete details.

# Making Code Readable

Commenting code and limiting line length both promote readability. See the following sections for details.

## Using Code Comments

In IDL, the semicolon (;) is the comment character. When IDL encounters the semicolon, it ignores the remainder of the line. It is good programming practice to fully document programs with comments. Comments in IDL do not slow down code execution or add noticeable size to IDL files.

A comment can exist on a line by itself, or can follow another IDL statement, as shown below:

```
; This is a comment
COUNT = 5    ; Set the variable COUNT equal to 5.
```

## Using Line Continuations

The line continuation character ($) allows you to break a single IDL statement into multiple lines. The dollar sign at the end of a line indicates that the current statement is continued on the following line. The dollar sign character can appear anywhere a space is legal, except within a string constant or between a function name and the first open parenthesis. Any number of continuation lines are allowed. The following example shows a line continuation after the space at the end of the third line:

```
PRO sample_recurse2, oNode, indent
   ;; "Visit" the node by printing its name and value
   PRINT, indent gt 0 ? STRJOIN(REPLICATE(' ', indent)) : '', $
      oNode->GetNodeName(), ':', oNode->GetNodeValue()
...
```

# Command Line Tips and Tricks

Entering text at the command line allows you to perform ad hoc analysis, compile and launch applications, and create $MAIN$ programs. IDL provides some valuable command line functionality to support these tasks. See the following sections for details.

**Note** ─────────────────────────────────────
─────────────────────────────────────────────

## Copying and Pasting Multiple IDL Code Lines

You can paste multiple lines of text from the clipboard to the command line. You simply need to place some text in the clipboard and paste it into the command line. Any source of text is valid, with emphasis on the requirement that the text be convertible to ASCII.

When you are using the tty-based command line version of IDL and you paste multiple lines, make sure that they contain only a single IDL command or are statements containing line continuation characters ($). Multi-line statements will produce unintended IDL interpreter behavior or errors. Lines are transferred to the command line as is. Namely, leading white space is not removed and comment lines are sent to the IDL interpreter without distinction.

If you are using the IDL Workbench, you can paste multiple statements directly into the Command Line view. You can also drag single or multiple lines from the Command History view to the Command Line view.

# Recalling Commands

By default, IDL saves the last 500 commands entered in a *recall buffer*. These command lines can be viewed, edited, and re-entered. The Command History view to the right of the Console displays the command history, organized by day.

You can re-use and edit commands by recalling them on the Command Line. The up-arrow key ( ↑ ) on the keypad recalls the previous command you entered to IDL. Pressing it again recalls the previous line, moving backward through the command history list. The down-arrow key ( ↓ ) on the keypad moves forward through the command history.

**Note** ────────────────────────────────────────────────────────────────
Using the HELP procedure with the RECALL_COMMANDS keyword displays the entire contents of the recall buffer in the IDL Console.
────────────────────────────────────────────────────────────────────────

Command recall is always available in the IDL Workbench. The command recall feature is enabled for the tty-based command-line version of IDL by setting the IDL_EDIT_INPUT preference to true, which sets the system variable !EDIT_INPUT to a non-zero value (the default is 1). See "!EDIT_INPUT" (Appendix D, *IDL Reference Guide*) for details.

## Changing the Number of Lines Saved

You can change the number of command lines saved in the recall buffer by setting the IDL_RBUF_SIZE preference equal to a number other than one (in the IDL Workbench, you can set this value via the **General** tab of the IDL Workbench Preferences dialog as well.)

# Special Command Line Characters

Commands entered at the IDL prompt are usually interpreted as IDL statements to be executed. Other interpretations include executive commands that control execution and compilation of programs, shell commands, and so on. Input to the IDL prompt is interpreted according to the first character of the line, as shown in the following table.

**Note** ———————————————————————————————————————

The information in this section applies equally to IDL used in command-line mode or in the IDL Workbench.

| First Character | Action |
|---|---|
| . | Executive command. See "About Executive Commands" on page 38 for details. |
| ? | Help inquiry. For example, enter ? on the Command Line to open the online help system. Enter ? .RUN to open the Help system to the page that explains the .RUN command. |
| $ | Send an operating system commands to a subprocess.<br>**Note -** the SPAWN procedure is a more flexible alternative. It need not be used interactively and the standard output of the command can be saved in an IDL string array. See "SPAWN" (*IDL Reference Guide*) for details. |
| @ | Batch file initiation. |
| ↑ or ↓ key | Recall/edit previous commands. |
| CTRL+D | In UNIX command-line mode, exits IDL, closes all files, and returns to operating system. |
| CTRL+Z | In UNIX command-line mode, suspends IDL. |
| All others | IDL statement. |

*Table 2-2: Interpretation of the First Character in an IDL Command*

### About Executive Commands

IDL executive commands compile programs, continue stopped programs, and start previously compiled programs. All of these commands begin with a period and must be entered in response to the IDL prompt. Commands can be entered in either uppercase or lowercase and can be abbreviated. Under UNIX, filenames are case sensitive; under Microsoft Windows, filenames can be specified in any case. See "Executive Commands" (*IDL Quick Reference*) for a descriptions of the available executive commands.

**Note**
Comments (prefaced by the semicolon character in IDL code) are not allowed within executive commands.

Executive commands are used to create $MAIN$ programs. See "Creating $MAIN$ Programs" on page 22 for details.

# Special Command Line Key Combination

When working at the command line, key combinations can be used to quickly edit a command. The line-editing abilities and the keys that activate them differ somewhat between the different operating systems. To access the history of commands entered at the command line, see "Recalling Commands" on page 36.

**Note**
The behavior can also differ within the same operating system, between the terminal window-based command line version of IDL and the Command Line view in the IDL Workbench.

The table below lists the edit functions and the corresponding keys in the terminal window (Command Line) and Workbench versions of IDL.

| Function | IDL Command Line | IDL Workbench |
|---|---|---|
| Move cursor to start of line | CTRL+A or Home | CTRL+A or Home |
| Move cursor to end of line | CTRL+E or End | CTRL+A or Home |
| Move cursor left one character | Left arrow | Left arrow |

*Table 2-3: Command Recall and Line Editing Keys*

| Function | IDL Command Line | IDL Workbench |
|---|---|---|
| Move cursor right one character | Right arrow | Right arrow |
| Move cursor left one word | CTRL+B, (R13 on Sun Keyboard) | CTRL+left arrow |
| Move cursor right one word | CTRL+F, (R15 on Sun Keyboard) | CTRL+right arrow |
| Delete from current to start of line | CTRL+U | CTRL+U |
| Delete from current to end of line | CTRL+K | CTRL+K |
| Delete current character | CTRL+X or CTRL+D | CTRL+X or Delete |
| Delete previous character | CTRL+H, or Backspace, or Delete | Backspace |
| Delete previous word | CTRL+W, or ESC-Delete | |
| Generate IDL keyboard interrupt | CTRL+C | CTRL+C (when a program is running) |
| Move back one line in recall buffer | CTRL+N, Up arrow | Up arrow |
| Move forward one line in recall buffer | Down arrow | Down arrow |
| Redraw current line | CTRL+R | |
| Overstrike/Insert | ESC-I | |
| EOF if current line is empty, else EOL | CTRL+D | |
| Search recall buffer for text | Available only in command-line mode. Enter ^ , then input *search string* at prompt. | |

*Table 2-3: Command Recall and Line Editing Keys (Continued)*

| Function | IDL Command Line | IDL Workbench |
|---|---|---|
| Tab command matching | Available only for Windows.<br><br>After entering a few characters, press Tab to cycle through all the matching commands in the command history. | After typing a few characters, press Tab to cycle through all the matching commands in the command history. |
| Insert the character at the current Executive Commands position | any character | any character |
| Parenthesis matching | | Place cursor outside of parenthesis and IDL highlights the matching parenthesis. |

*Table 2-3: Command Recall and Line Editing Keys (Continued)*

# Recording IDL Command Line Input

Journaling provides a record of an interactive session by saving all text entered from the Command Line in a file. In journaling, all text entered to the IDL prompt is entered directly into the file, and any text entered from the terminal in response to any other input request (such as with the READ procedure) is entered as a comment. The result is a file that contains a complete description of the IDL session. JOURNAL has the form:

```
JOURNAL[, Argument]
```

where *Argument* is either a filename (if journaling is not currently in progress) or an expression to be written to the file (if journaling is active). The first call to JOURNAL starts the logging process. If no argument is supplied, a journal file named idlsave.pro is started.

**Warning** ───────────────────────────────────────────────
Under all operating systems, creating a new journal file will cause any existing file with the same name to be lost. Supply a filename argument to JOURNAL to avoid destroying existing files.
────────────────────────────────────────────────────────

When journaling is not in progress, the value of the system variable !JOURNAL is zero. When the journal file is opened, the value of this system variable is set to the number of the logical file unit on which the file is opened. This allows IDL routines to check if journaling is active. You can send any arbitrary data to this file using the normal IDL output routines. In addition, calling JOURNAL with an argument while journaling is in progress results in the argument being written to the journal file as if the PRINT procedure had been used. In other words, the statement,

```
JOURNAL,
```

is equivalent to

```
PRINTF, !JOURNAL, Argument
```

with one significant difference—the JOURNAL statement is not logged to the file, only its output; while the PRINTF statement will be logged to the file in addition to its output.

Journaling ends when the JOURNAL procedure is called again without an argument or when IDL is exited. The resulting file serves as a record of the interactive session that went on while journaling was active. It can be used later as an IDL batch input file to repeat the session, and it can be edited with any text editor if changes are necessary. See "JOURNAL" (*IDL Reference Guide*) for examples.

# Interrupting or Aborting Execution

To manually stop programs that are running, issue a *keyboard interrupt* by typing **Ctrl+C**.

**Note** ─────────────────────────────────────────────────────────────

If you are using the IDL Workbench, **Ctrl+C** interrupts execution only when the focus is in the IDL Command Line view and a program is running. In other situations, **Ctrl+C** copies the current selection to the clipboard.

─────────────────────────────────────────────────────────────────────

A message indicating the statement number and program unit being executed is issued on the terminal or IDL Console acknowledging the interrupt. The values of variables can be examined, statements can be entered from the keyboard, and variables can be changed. The program can be resumed by issuing the .CONTINUE executive command to resume or the .STEP executive command to execute the next statement and stop.

## Variable Context After Interruption

When a program is interrupted, the variable context is within the program unit where the program stopped. IDL checks for interrupts after each statement. Program execution does not stop until the active statement finishes, so it can take some time after you type an interrupt for the program to be interrupted.

**Note** ─────────────────────────────────────────────────────────────

You can view the variables in a program using the IDL Workbench Variables view.

─────────────────────────────────────────────────────────────────────

To revert to the next-higher program level, use the RETURN statement at the Command Line. You can repeat this command until the program returns to the main level. To return control to the main program level, use the RETALL command. To find out where the interrupt occurred, use the HELP command to determine the program context.

Varibles view

## Aborting IDL on UNIX Systems

If you use IDL in command-line mode on a UNIX system and need to abort rather than exit using the EXIT command, type **Ctrl+\**. This is a very abrupt exit—all variables are lost, and open files may not be saved. You should always close IDL

using the EXIT command when possible. Avoid using **Ctrl**+\ except in emergency situations.

**Note** ───────────────────────────────────────────────

After aborting IDL by using **Ctrl**+\, you may find that your terminal is left in the wrong state. You can restore your terminal to the correct state by issuing one of the following UNIX commands:

```
% reset   or % stty echo -cbreak
```

─────────────────────────────────────────────────────────────

# For More Information on Programming

Here we have just touched on the possibilities that IDL offers for programmers. For more information on how to prepare and run programs, see Chapter 5, "Creating Procedures and Functions" for creating and calling procedures and functions. It also describes argument and keyword parameters, and keyword inheritance.

# Chapter 3
# Executing Batch Jobs in IDL

The following topics are covered in this chapter:

# Overview of Batch Files

A batch file contains one or more IDL statements or commands. Each line of the batch file is read and executed before proceeding to the next line. This makes batch files different from main-level programs, which are compiled as a unit before being executed, and named programs, in which all program modules are compiled as an unit before being executed. A file created by the JOURNAL routine is an example of an batch file. Program types and more information on journaling are described in Chapter 2, "Creating and Running Programs in IDL".

**Note** ───────────────────────────────────────────────────────────────

Batch files are sometimes referred to as *include files*, since they can be used to "include" the multiple IDL statements contained in the file in another file.

───────────────────────────────────────────────────────────────────────

See the following topics for more information on batch files:

- "Batch File Execution" on page 47
- "Interpretation of Batch Statements" on page 49
- "A Batch Example" on page 50

**Tip** ────────────────────────────────────────────────────────────────

For information on how to specify a batch file as a startup file that is automatically executed when IDL is started, see "Startup File" (Chapter 1, *Using IDL*).

───────────────────────────────────────────────────────────────────────

# Batch File Execution

You can run IDL in non-interactive mode (batch mode) by entering the character @ followed by the name of a file containing IDL executive commands and statements. Commands and statements are executed in the order they are contained in the file, as if they had been entered at the IDL command prompt.

Batch execution can be terminated before the end of the file, with control returning to interactive mode without exiting IDL, by calling the STOP procedure from the batch file. Calling the EXIT procedure from the batch procedure has the usual effect of terminating IDL.

## Executing a Batch File

To execute a batch file, enter the name of the file, prefaced with the "@" character, at the IDL prompt:

```
@batchfile
```

where *batchfile* is the name of the file containing IDL statements. Note that the @ symbol must be the first character on the line in order for it to be interpreted properly.

**Note** ───────────────────────────────────────────────────────────────
   This syntax can also be used within an IDL program file.
─────────────────────────────────────────────────────────────────────────

The cntour01 batch file contains the following lines:

```
; Restore Maroon Bells data into the IDL variable "elev".
RESTORE, FILEPATH('marbells.dat', SUBDIR=['examples','data'])
; Make the x and y vectors giving the column and row positions.
X = 326.850 + .030 * FINDGEN(72)
Y = 4318.500 + .030 * FINDGEN(92).
```

Enter the following at the IDL command line to execute the batch file:

```
@cntour01
```

IDL reads statements from the specified file until the end of the file is reached. Variables *ELEV*, *X*, and *Y* appear in the variable watch window. Batch files can also be nested by placing a call to one batch file within another. For example, the surf01 batch file calls the cntour01 batch file and uses the variable data to create a surface display. To see the results, enter the following at the command line:

```
@surf01
```

# Naming and Locating Batch Files

If *filename* does not include a file extension, IDL searches the current working directory and the directories specified by the !PATH system variable for a file with *filename* as its base, with the file extension .pro. If *filename*.pro is not found in a given directory, IDL searches for *filename* with no extension in that directory. If *filename* is found (with or without the .pro extension), the file is executed and the search ends. If *filename* includes a full path specification, IDL does not search the directories in !PATH.

# Interpretation of Batch Statements

Each line of a batch file is interpreted exactly as if it was entered from the keyboard. In batch mode, IDL compiles and executes each statement before reading the next statement. This differs from the interpretation of main-level programs compiled using .RNEW or .RUN, in which all statements in a program are compiled as a single unit and then executed.

GOTO statements are illegal in the batch mode because each batch file statement is compiled and executed sequentially.

Multiline statements must be continued on the next line using the $ continuation character, because IDL terminates every interactive mode statement not ending with $ by an END statement. A common mistake is to include a multiple-line block statement in a batch file as shown below.

```
; This will not work in batch mode.
FOR I = 1, 10 DO BEGIN
   A = X[I]
   ...
   ...
ENDFOR
```

In batch mode, IDL compiles and executes each line separately, causing syntax errors in the above example because no matching ENDFOR is found on the line containing the BEGIN statement when the line is compiled. The above example could be made to work by writing the block of statements as a single line using the $ (continuation) and & (multiple commands on a single line) characters.

# A Batch Example

You can create a batch file in the IDL Editor or other text editor program. An example of an IDL executive command line that initiates batch execution:

```
@myfile
```

This command causes the file `myfile` to be used for statement and command input. If this file is not in the current directory, the directories specified by !PATH are also searched.

An example of the contents of a batch file follows:

```
; Run program A:
.RUN proga
; Run program B:
.RUN progb
; Print results:
PRINT, AVALUE, BVALUE
; Close unit 3:
CLOSE, 3
```

The batch file should not contain complete program units. Complete program units should be compiled and run by using the .RUN and .RNEW commands in the batch files, as illustrated above.

### Example Code

Several working batch files are included in the distribution. For an example, type `@sigprc09` at the IDL prompt to run the batch file. The source code for this example is located in `sigprc09`, in the `examples/doc/signal` directory.

# Chapter 4
# Creating SAVE Files of Programs and Data

The following topics are covered in this chapter:

# Overview of SAVE Files

You can create binary files containing data variables, system variables, functions, procedures, or objects using the SAVE procedure. These SAVE files can be shared with other users who will be able to execute the program, but who will not have access to the IDL code that created it. Variables that are used from session to session can be saved as and recovered from a SAVE file.

**Tip** ────────────────────────────────────────────────────────────────
A startup file can be set up to execute the RESTORE command every time IDL is started. See "Startup File" (Chapter 1, *Using IDL*) for information on specifying a startup files.
────────────────────────────────────────────────────────────────────────

**Note** ───────────────────────────────────────────────────────────────
Files containing IDL routines and system variables can only be restored by versions of IDL that share the same internal code representation. Since the internal code representation changes regularly, you should always archive the IDL language source files (`.pro` files) for routines you are placing in IDL SAVE files so you can recompile the code when a new version of IDL is released.
────────────────────────────────────────────────────────────────────────

## What Can be Stored in a SAVE File

A SAVE file can contain system variables, data variables, or named program files. See the following topics for details:

- Named routines — store one or more routines in a single SAVE file and distribute it other IDL users. See "About Program and Data SAVE Files" on page 54.

- Variable data — store system or session variable data in a SAVE file. See "Saving Variables from an IDL Session" on page 65.

**Warning** ─────────────────────────────────────────────────────────────
Variables and routines cannot be stored in the same SAVE file.
────────────────────────────────────────────────────────────────────────

## Save Files and Application Development

For distributable applications, IDL does not compile `.pro` files. Therefore, any procedures or functions used by an application must be resolved and contained in a SAVE file. For IDL applications, these routines can be part of the main SAVE file

that is restored when your application is started. The following are examples of cases in which you might use SAVE to create .sav files:

- To create SAVE files for any procedures or functions that are not contained in the main SAVE file that is restored when a native IDL application is started

- To create SAVE files for any procedures or functions used by a Callable IDL or ActiveX application

- To create SAVE files for any variables used by your application, such as custom ASCII templates

If your application is composed of a number of procedures and other types of files, it would likely be easier to create a SAVE file using the IDL Workbench *Build Project* interface; see Running and Building IDL Projects for information. See Chapter 23, "Distributing Runtime Mode Applications" for more information on creating applications in IDL, including how to license your application and package it for distribution.

# Accessing and Running SAVE Files

Depending upon the name and contents of the SAVE file, there are a number of ways to restore the file. SAVE files containing routines can be executed in a fully licensed version of IDL, through the IDL Virtual Machine (if created in IDL version 6.0 or later), or using the IDL_Savefile object. SAVE files containing variable data can be restored using the RESTORE procedure or the IDL_Savefile object. You may also be able to automatically compile and restore the file by typing the name of the file at the command line. See "Executing SAVE Files" on page 67 for details.

# About Program and Data SAVE Files

The SAVE procedure can be used to quickly save IDL routines and data variables in a binary format that can be shared with other IDL users, or with others who have installed the IDL Virtual Machine. If you are developing an application for distribution to users who do not have a version of IDL installed, you should also see Chapter 23, "Distributing Runtime Mode Applications".

**Warning** —————————————————————————————————————————
Variables and routines cannot be stored in the same SAVE file.

**Note** ———————————————————————————————————————————
While IDL routines or data can be saved in a file with any extension, it is common to use the extension .sav for SAVE files. Using the .sav extension has two benefits: it makes it clear to another IDL user that the file contains IDL routines or data, and it allows IDL automatically locate and compile the routines in the file as described in "Automatic Compilation" on page 30.

If your program or utility consists of multiple routines, each procedure or function used by your program must be resolved and contained in a SAVE file. You have the following options:

- Include all routines in a main SAVE file that is restored first. This makes all routines available without having to restore any additional SAVE files. You can do this manually, by compiling all of the routines yourself (possibly with the assistance of the RESOLVE_ALL or IRESOLVE routines).

- Create a separate SAVE file for each routine used by your application. Assuming each SAVE file uses the .sav extension and has the same name as the procedure or function it contains, this allows you to simply place the files in a directory included in !PATH; IDL will compile all of the files automatically when needed.

If your program also contains variable data, you must create a separate SAVE file to contain the data. Variable data must be explicitly restored before any routine attempts to use the variables contained in the file. See "Executing SAVE Files" on page 67 for more information.

**Note** ───────────────────────────────────────

A SAVE file containing data will always be restorable. However, SAVE files created prior to IDL version 5.5 that contain IDL procedures, functions, and programs are not always portable between different versions of IDL. If you created your SAVE file with a version of IDL earlier than 5.5, you will need to recompile your original .pro files and re-create the SAVE file using the current version of IDL.

# Creating SAVE Files of Program Files

The following examples create SAVE files that are stand-alone IDL applications that can be run on any Windows, UNIX or Mac OS X computer containing the IDL Virtual Machine or a licensed copy of IDL. See the following examples:

- Example: A SAVE File of a Simple Routine below creates two SAVE files. One SAVE file contains variable data, the other SAVE file contains a procedure uses RESTORE to access the variable data in the first SAVE file.

- "Example: A Save File of a Simple Widget Application" on page 59 displays an image in a simple widget application.

- "Example: Creating a SAVE File of an Object Definition" on page 60 shows the special steps that must be taken when creating a SAVE file of an object that has dependencies upon other objects.

- "Example: A SAVE File of a Custom iPlot Display" on page 62 restores variable data and plots it in an iPlot display.

**Note** ─────────────────────────────────────────────────────
If you want your customers to run your application on a computer without IDL, you will need to include a runtime version of IDL with a runtime or embedded license in your application distribution. See Chapter 23, "Distributing Runtime Mode Applications" for details.
─────────────────────────────────────────────────────────────

# Example: A SAVE File of a Simple Routine

The following example creates two SAVE files. One SAVE file contains variable data, loaded from an image file. This SAVE file is then restored by the program in the main SAVE file, which uses a simple call to the ARROW procedure to point out an area of interest within the image.

## Save Image Variable Data

1. **Start a fresh session of IDL.** This avoids saving unwanted session information.

2. **Read image data into a variable.** Open an image file containing an MRI proton density scan of a human thorax, and read the data into a variable named image*:*

   ```
   READ_JPEG, (FILEPATH('pdthorax124.jpg', SUBDIRECTORY= $
       ['examples', 'data'])), image
   ```

3. **Create a SAVE file containing the image data.** Use the SAVE procedure to save the *image* variable in a SAVE file by entering the following:

```
SAVE, image, FILENAME='imagefile.sav'
```

This stores the SAVE file in your current working directory.

**Note** ───────────────────────────────────────────

When using the SAVE procedure, some users identify binary files containing variable data using a `.dat` extension instead of a `.sav` extension. While any extension can be used to identify files created with SAVE, it is recommended that you use the `.sav` extension to easily identify files that can be restored.

─────────────────────────────────────────────────

## Save a Procedure that Restores Variable Data

1. **Create the program file.** Create the following IDL program that first restores the *image* variable contained within the `imagefile.sav` file. This variable is used in the following program statements defining the size of the window and in the TV routine which displays the image. The ARROW routine then draws an arrow within the window. Enter the following lines in a text editor.

```
PRO draw_arrow

; Restore image data.
RESTORE, 'imagefile.sav'

; Get the dimensions of the image file.
s = SIZE(image, /DIMENSIONS)

; Prepare display device and display image.
DEVICE, DECOMPOSED = 0
WINDOW, 0, XSIZE=s[0], YSIZE=s[1], TITLE="Point of Interest"
TV, image

; Draw the arrow.
ARROW, 40, 20, 165, 115

; The IDL Virtual Manchine exits IDL when the end of a
; program is reached if there are not internal events. The
; WAIT statement here allows the user to view the .sav file
; results for 10 seconds when executed through the IDL
; Virtual Machine.
WAIT, 10

END
```

2. **Save the file.** Name the saved file draw_arrow.pro.

3. **Reset the IDL session.** Enter the following at the IDL prompt to ensure that no unwanted session information is saved along with the program:

   ```
   .FULL_RESET_SESSION
   ```

4. **Compile the program.** Enter the following at the IDL prompt:

   ```
   .COMPILE draw_arrow
   ```

5. **Resolve dependencies.** Use RESOLVE_ALL (or IRESOLVE if the routine has any dependencies on iTools components) to iteratively compile any uncompiled user-written or library procedures or functions that are called in any already-compiled procedure or function:

   ```
   RESOLVE_ALL
   ```

   **Note** ───────────────────────────────────────────────

   RESOLVE_ALL does not resolve procedures or functions that are called via quoted strings such as CALL_PROCEDURE, CALL_FUNCTION, or EXECUTE, or in keywords that can contain procedure names such as TICKFORMAT or EVENT_PRO. You must manually compile these routines.

   ─────────────────────────────────────────────────────────

6. **Create the SAVE file.** Create a file called draw_arrow.sav that contains the user-defined draw_arrow procedure. When the SAVE procedure is called with the ROUTINES keyword and no arguments, it create a SAVE file containing all currently compiled routines. Because the procedures within the draw_arrow procedure are the only routines that are currently compiled in the IDL session, create the SAVE file as follows:

   ```
   SAVE, /ROUTINES, FILENAME='draw_arrow.sav'
   ```

   **Note** ───────────────────────────────────────────────

   When the name of the SAVE file uses the .sav extension and has the same base name as the main level program, it can be automatically compiled by IDL. This means that it can be called from another routine or restored from the IDL command line using only the name of the saved routine. See "Automatic Compilation" on page 30 for details.

   ─────────────────────────────────────────────────────────

7. **Test the SAVE file**. Select **Start → Programs → IDL 7.1 → IDL Virtual Machine**. Click on the splash screen and open draw_arrow.sav. You could also test the SAVE file from IDL, enter the following at the command prompt.

   ```
   RESTORE, 'draw_arrow.sav'
   draw_arrow
   ```

See "Executing SAVE Files" on page 67 for all the available ways to run a SAVE file.

# Example: A Save File of a Simple Widget Application

The following example creates a native IDL application that displays an image in a simple widget interface. When any application runs in the IDL Virtual Machine, there must some element (such as widget or interface events, or a WAIT statement) that keeps the application from immediately exiting with the END statement is reached. This example includes a Done button for this reason. The example in "Example: A SAVE File of a Simple Routine" on page 56 includes a WAIT statement.

1. **Create a .pro file.** Enter the following in the IDL Editor, and save it as myApp.pro:

```
PRO done_event, ev
; When the 'Done' button is pressed, exit
; the application.

WIDGET_CONTROL, ev.TOP, /DESTROY

END

PRO myApp

; Read an image file.
READ_JPEG, (FILEPATH('endocell.jpg', SUBDIRECTORY = $
   ['examples', 'data'])), image

; Find the dimensions of the image.
info = SIZE(image,/DIMENSIONS)
xdim = info[0]
ydim = info[1]

; Create a base widget containing a draw widget
; and a 'Done' button.
wBase = WIDGET_BASE(/COLUMN)
wDraw = WIDGET_DRAW(wBase, XSIZE=xdim, YSIZE=ydim)
wButton = WIDGET_BUTTON(wBase, VALUE='Done',
EVENT_PRO='done_event')

; Realize the widgets.
WIDGET_CONTROL, wBase, /REALIZE

; Retrieve the widget ID of the draw widget.
WIDGET_CONTROL, wDraw, GET_VALUE=index
```

```
; Set the current drawable area to the draw widget.
WSET, index

; Display some data.
TV, image

; Call XMANAGER to manage the event loop.
XMANAGER, 'myApp', wBase, /NO_BLOCK

END
```

2. **Reset the IDL session.** Enter the following at the IDL prompt to ensure that no unwanted session information is saved along with the program:

   ```
   .FULL_RESET_SESSION
   ```

3. **Compile the application**. Select **Run → Compile** to compile the .pro file.

4. **Resolve dependencies.** Type RESOLVE_ALL at the command line to resolve all procedures and functions that are called in the application:

   ```
   RESOLVE_ALL
   ```

   **Note** ────────────────────────────────────────────────────

   If your program relies on iTools components, use IRESOLVE instead of RESOLVE_ALL.

   ────────────────────────────────────────────────────────────

5. **Create the SAVE file.** Enter the following to save the compiled application as a SAVE file:

   ```
   SAVE, /ROUTINES, FILENAME = 'myApp.sav'
   ```

See "Executing SAVE Files" for ways to run the SAVE file.

# Example: Creating a SAVE File of an Object Definition

When you create a SAVE file that contains an object defined in a .pro file, you must save the .pro file as a SAVE file, just like any other procedure you wish to distribute. However, it is important to note that if the object has any inherited properties from superclasses or other objects, and the object definitions exist in .pro files, you must also compile and include these object definition files in your SAVE file. Objects using a .pro extension typically exist in the IDL distribution's lib subdirectory and its subdirectories.

**Note**

Do not confuse the process of saving an *instance* of an object with saving its *definition*. A reference to an instantiated object is stored in an IDL variable, and must be saved in a SAVE file as a variable. An object definition, on the other hand, is an IDL routine, and must be saved in a SAVE file as a routine. *It is important to remember that restoring an instance of an object does not restore the object's definition.* If the object is defined in `.pro` code, you must save the object definition routine and the object instance variable in separate SAVE files, and restore both instance and definition in the target IDL session.

The IDL distribution includes and example of a composite object composed of an image, a surface, and a contour, which are combined into a single object called the IDLexShow3 object. To see this object being used in an application, run the `show3_track.pro` file in the `examples/doc/objects` directory. This procedure has dependencies on two objects (`trackball.pro` and `IDLexShow3__define.pro`). You must use RESOLVE_ALL and explicitly include these two objects in the CLASS keyword string array in order to create a valid SAVE file.

If you fail to resolve all object dependencies, you will receive an error stating that there was an attempt to call and undefined procedure or function when you run the SAVE file. If the error references an object, add the object name to the CLASS keyword string array to resolve the problem. Undefined procedure or function errors are more likely to appear when you restore a SAVE file using the IDL Virtual Machine, which does not search !PATH to resolve routines. Using RESTORE at the command line does search !PATH. Therefore, a SAVE file that can be successfully executed using RESTORE may not succeed when called from the IDL Virtual Machine. If you are distributing SAVE files to users running the IDL Virtual Machine, make sure to test the SAVE file in the Virtual Machine.

Complete the following steps to create a save file of an object:

1.  **Reset your session.** Either start a new IDL session or enter the following at the IDL prompt to ensure that no unwanted session information is saved along with the program:

    ```
    .FULL_RESET_SESSION
    ```

2.  **Open the main procedure.** Open and compile `show3_track.pro` file by entering the following at the IDL command prompt:

    ```
    .COMPILE Show3_Track.pro
    ```

3. **Resolve object dependencies.** Use the CLASS keyword to resolve dependencies to other object .pro files by passing it a string or string array containing the name(s) of the objects:

   ```
   RESOLVE_ALL, CLASS=['Trackball', 'IDLexShow3']
   ```

4. **Create the SAVE file.** Enter the following at the IDL command prompt:

   ```
   SAVE, /ROUTINES, FILENAME='show3_track.sav'
   ```

5. **Test the SAVE file**. Select **Start → Programs → IDL 7.1 → IDL Virtual Machine**. Click on the splash screen and open show3_track.sav. You could also test the SAVE file from IDL. Enter the following at the command prompt.

   ```
   RESTORE, 'show3_track.sav'
   show3_track
   ```

See "Executing SAVE Files" on page 67 for all the available ways to run a SAVE file.

# Example: A SAVE File of a Custom iPlot Display

The following example configures a custom iPlot display and stores the program in a SAVE file. Restoring the SAVE file opens iPlot with the specified data.

**Note**
_____

When working with iTools, you can create an iTool State (.isv) file that contains data and application state information.You can share this file with other IDL users who have the same version or a newer version of IDL. See the *iTool User's Guide* for details. This is not the same as packaging iTools functionality into a SAVE file, which is described in this example. When iTools functionality is packaged into a SAVE file, it can be accessed by IDL users or through the IDL Virtual Machine.

_____

1. **Access and save data.** Save variable data from a batch file into a SAVE file:

   ```
   @plot01
   SAVE, FILENAME='plotdata01.sav'
   ```

2. **Create the program file.** This program restores data, and creates a plot display in an iPlot display. Enter the following lines in a text editor:

   ```
   PRO ex_saveiplot

   ; Define variables.
   RESTORE, 'plotdata01.sav'

   ; Use the LINFIT function to fit the data to a line:
   coeff = LINFIT(YEAR, SOCKEYE)
   ```

```
;YFIT is the fitted line:
YFIT = coeff[0] + coeff[1]*YEAR

; Plot the original data points with PSYM = 4, for diamonds:
iPLOT, YEAR, SOCKEYE, /YNOZERO, SYM_INDEX = 4, $
   SYM_COLOR=[255,0,0], LINESTYLE=6, $
   TITLE = 'Quadratic Fit', XTITLE = 'Year', $
   YTITLE = 'Sockeye Population'

; Overplot the smooth curve using a plain line:
iPLOT, YEAR, YFIT, /OVERPLOT

END
```

3. **Reset you session.** Enter the following at the IDL prompt to ensure that no unwanted session information is saved along with the program:

   ```
   .FULL_RESET_SESSION
   ```

4. **Compile the program.** Use the .COMPILE executive command as follows: Compile the main program file:

   ```
   .COMPILE ex_saveiplot
   ```

5. **Resolve dependencies.** Use IRESOLVE to resolve dependencies upon iTool components:

   ```
   IRESOLVE
   ```

6. **Create the SAVE file.** Use the /ROUTINES keyword to include all currently compiled routines:

   ```
   SAVE, /ROUTINES, FILENAME='ex_saveiplot.sav'
   ```

7. **Test the SAVE file**. Select **Start → Programs → IDL 7.1 → IDL Virtual Machine**. Click on the splash screen and open ex_saveiplot.sav. You could also run the SAVE file from IDL. Enter the following at the command prompt.

   ```
   RESTORE, 'ex_saveiplot.sav'
   ex_saveiplot
   ```

# Other Examples of SAVE File Creation

See the following topics for additional SAVE file examples:

- "ASCII_TEMPLATE" (*IDL Reference Guide*) contains Example: Create a SAVE File of a Custom ASCII Template

- "XROI" (*IDL Reference Guide*) contains the following SAVE file examples:

  - "Example: Save ROI Data"

  - "Example: Save the XROI Utility with ROI Data"

# Saving Variables from an IDL Session

In addition to distributing IDL code in binary format, you can also create SAVE files that contain variable data. The state of variables in an IDL session can be saved quickly and easily, and can be restored to the same point. This feature allows you to stop work, and later resume at a convenient time. Variables that you may wish to create a SAVE file of include frequently used data files or system variable definitions.

## Saving Data Variables in a SAVE File

Data can be conveniently stored in SAVE files, relieving you of the need to remember the dimensions of arrays and other details. It is very convenient to store images this way. For instance, if the three variables *Red*, *Green*, and *Blue* hold the color table vectors, and the variable *Image* holds the image variable, the IDL statement,

```
SAVE, FILENAME = 'image.sav', Red, Green, Blue, Image
```

will save everything required to display the image properly in a file named image.sav. At a later date, the simple command,

```
RESTORE, 'image.sav'
```

will recover the four variables from the file. See "Save Image Variable Data" on page 56 for an additional example.

## Saving Heap Variables in a SAVE File

The SAVE procedure works for heap variables just as it works for all other supported types. By default, when IDL saves a pointer or object reference in a SAVE file, it recursively saves the heap variables that are referenced by that pointer or object reference.

In some cases, you may want to save the pointer or object reference, but *not* the heap variable that are referenced by that pointer or object reference. You can specify that the heap variable associated with a pointer or object reference not be saved using the HEAP_NOSAVE procedure or the HEAP_SAVE function. See the documentation for HEAP_SAVE for additional details.

## Saving System Variables in a SAVE File

System variables can also be saved and later applied to another session of IDL. For instance, you may choose to customize !PATH, the system variable defining the directories IDL will search for libraries, batch/include files, and executive commands

or !P, the system variable that controls the definition of graphic elements associated with plot procedures. You can save these definitions in a SAVE file and later automatically restore or selectively restore the variables to apply the settings to other IDL sessions.

To save and restore the state of all current and system variables within an IDL session, you could use the following statement:

```
SAVE, /ALL, FILENAME = 'myIDLsession.sav'
```

The ALL keyword saves all system variables and local variables from the current IDL session. See Chapter 13, "Working with Data in IDL" for information on these elements of an IDL session.

**Note** ────────────────────────────────────────────────────

Routines and variables cannot be saved in the same file. Setting the ALL keyword does not save routines.

────────────────────────────────────────────────────────────

To restore the session information, enter:

```
RESTORE, 'myIDLsession.sav'
```

**Note** ────────────────────────────────────────────────────

If the file is not located in your current working directory, you will need to define the path to the file.

────────────────────────────────────────────────────────────

Long iterative jobs can save their partial results in a SAVE format to guard against losing data if some unexpected event such as a machine crash should occur.

**Note** ────────────────────────────────────────────────────

A SAVE file containing data will always be restorable. However, SAVE files created prior to IDL version 5.5 that contain IDL procedures, functions, and programs are not always portable between different versions of IDL. If you created your SAVE file with a version of IDL earlier than 5.5, you will need to recompile your original .pro files and re-create the SAVE file using the current version of IDL.

────────────────────────────────────────────────────────────

# Executing SAVE Files

IDL SAVE files (created using the SAVE procedure) can contain one or more routines that have been packaged into a single binary file. SAVE files can also contain system or data variables.

**Note** ──────────────────────────────────────────────────────
While IDL routines or data can be saved in a file with any extension, it is common to use the extension `.sav` for SAVE files. Using the `.sav` extension has two benefits: it makes it clear to another IDL user that the file contains IDL routines or data, and it allows IDL to locate routines with the same base name as the file in SAVE files located in IDL's path.
────────────────────────────────────────────────────────────

This section describes various ways to restore files created with the SAVE procedure. In order of increasing complexity and flexibility, your options are:

- "Using the IDL Virtual Machine to Run SAVE Files", described below
- "Executing SAVE Files in the IDL Workbench" on page 67
- "Executing SAVE Files From the Operating System" on page 68
- "Executing SAVE Files by Name" on page 68
- "Using RESTORE to Access SAVE Files" on page 69
- "Using the IDL_Savefile Object to Access SAVE Files" on page 70

## Using the IDL Virtual Machine to Run SAVE Files

Users without an IDL license can use the IDL Virtual Machine to access programs contained in SAVE files created in IDL version 6.0 or later. See "Starting a Virtual Machine Application" (Chapter 24, *Application Programming*) for instructions.

**Note** ──────────────────────────────────────────────────────
There are a few limitations to SAVE file contents discussed in "Limitations of Virtual Machine Applications" (Chapter 24, *Application Programming*).
────────────────────────────────────────────────────────────

## Executing SAVE Files in the IDL Workbench

If you use the IDL Workbench **File → Open** menu item to select a SAVE file with the extension `.sav`, IDL will automatically restore the file, with the following results:

- If the SAVE file contains routines, the routines are restored and IDL attempts to run a routine with the same name as the `.sav` file.

- If the SAVE file contains data, the variables are restored in IDL's $MAIN$ scope.

Similarly, if the SAVE file is included in an open project, double-clicking on the file in the Project Explorer view will restore the file's contents as described above.

# Executing SAVE Files From the Operating System

On Windows and Macintosh systems, you can execute or restore a SAVE file directly from the operating system's file manager. Simply double-click on a file with the extension `.sav` in the Windows Explorer or Macintosh Finder.

If the SAVE file contains routines:

- If IDL is already running, the routines are restored and IDL attempts to run a routine with the same name as the `.sav` file.

- If IDL is not running, it launches in Virtual Machine mode, restores the routines and attempts to run a routine with the same name as the `.sav` file.

If the SAVE file contains data, the IDL Workbench launches automatically (if it is not already running) and the variables are restored in IDL's $MAIN$ scope.

# Executing SAVE Files by Name

You can execute a program stored in a SAVE file from the IDL command line by typing in the name of the routine if the file meets the following conditions:

- The SAVE file has the same base name as the routine you wish to run

- The SAVE file has the extension `.sav`

- The SAVE file is stored in a directory included in the !PATH system variable

Call the procedure with the same name as the `.sav` file to restore the program and execute it immediately using IDL's *automatic compilation* mechanism. IDL will search the current directory then the path specified by !PATH for a `.sav` file with the name of the called routine and, if it finds the `.sav` file, it restores, compiles and executes it automatically.

For example, to restore and execute the `draw_arrow` routine contained in the file `draw_arrow.sav` (created in "Example: A SAVE File of a Simple Routine" on page 56), enter the following at the command line:

```
draw_arrow
```

IDL will search for a file named either draw_arrow.pro or draw_arrow.sav, beginning in the current working directory and then searching in each directory specified by !PATH. When it finds a file whose name matches (in this case, draw_arrow.sav), it will compile the routines in the file up to and including the routine whose name matches the filename. IDL then executes the routine with the matching name. See "Automatic Compilation" on page 30 for additional details.

# Using RESTORE to Access SAVE Files

Use the RESTORE procedure to explicitly restore the entire contents of a SAVE file that contains variable data or program files. Because calling a procedure with the same name as a SAVE file allows IDL to automatically find and restore the SAVE file, it isn't always necessary to explicitly restore a .sav file using RESTORE. Cases in which you *must* use RESTORE include the following:

- When you are restoring a SAVE file containing variable data.

- When your SAVE file contains multiple routines, and you need to first call a routine that uses a different name than the .sav file. For example, if you have a SAVE file named routines.sav that contains the ARROW and BAR_PLOT procedures, you would need to restore routines.sav before calling ARROW or BAR_PLOT.

Using RESTORE is more powerful and flexible than relying on IDL's rules for automatic compilation, for the following reasons:

- The restored SAVE file can contain IDL variable data

- If the restored SAVE file contains IDL routines, *all* routines contained in the file will be restored, and *none* will be executed

- The restored SAVE file can have any filename and extension

- The restored SAVE file can be located in any directory

For example, in "Example: A SAVE File of a Simple Routine" on page 56, we created two SAVE files: imagefile.sav and draw_arrow.sav. The imagefile.sav file contains image variable data. To restore the image data, enter the following at the IDL command line:

```
RESTORE, 'imagefile.sav'
```

IDL creates the variable *image* in the current scope using the saved variable data.

If the file you are attempting to restore is not located in your current working directory, you will need to specify a path to the file. RESTORE does not search for

SAVE files in any other directory. For example, if draw_arrow.sav is located in myappdir, restore it using the following statement:

```
RESTORE, 'myappdir/draw_arrow.sav'
```

# Using the IDL_Savefile Object to Access SAVE Files

You can use the IDL_Savefile object class to gain information about the contents of a SAVE file, and to selectively restore items from the save file. Once a routine has been restored via calls the IDL_Savefile object, you can execute it simply by typing its name at the IDL command prompt. For example, if an IDL program named myroutine is stored in myroutine.sav, which is located in a directory that is *not* in !PATH, entering the following at the IDL command line will restore the routine and execute it:

```
obj = OBJ_NEW('IDL_Savefile', 'path/myroutine.sav')
obj->RESTORE, 'myroutine'
myroutine
```

where *path* is the full path to the myroutine.sav file. See "Getting Information About SAVE Files" (Chapter 4, *Using IDL*) for additional details.

# Changes to IDL 5.4 SAVE Files

With IDL 5.4, IDL became 64-bit capable. The original IDL SAVE/RESTORE format used 32-bit offsets. In order to support 64-bit memory access, the IDL SAVE/RESTORE file format was modified to allow the use of 64-bit offsets within the file, while retaining the ability to read old files that use the 32-bit offsets.

The SAVE command always begins reading any SAVE file using 32-bit offsets. If the 64-bit offset command is detected, 64-bit offsets are then used for any subsequent commands.

- In IDL versions capable of writing large files (!VERSION.FILE_OFFSET_BITS EQ 64), SAVE writes a special command at the beginning of the file that switches the format from 32 to 64-bit.

- SAVE always starts reading any SAVE file using 32-bit offsets. If it sees the 64-bit offset command, it switches to 64-bit offsets for any commands following that one.

This configuration is fully backward compatible, in that any IDL program can read any SAVE file it has created, or by any earlier IDL version. Note however that files produced in IDL 5.4 using 64-bit offsets are not readable by older versions of IDL.

It has come to our attention that IDL users commonly transfer SAVE/RESTORE data files written by newer IDL versions to sites where they are restored by older versions of IDL. It is not generally reasonable to expect this sort of forward compatibility, and it does not fit the usual definition of backwards compatibility. We have always strived to maintain this compatibility. However, in IDL 5.4 this was not the case. The following steps were taken in IDL 5.5 to minimize the problems caused by the IDL 5.4 save format:

- 64-bit offsets encoding has been improved. SAVE files written by IDL 5.5 and later should be readable by any previous version of IDL, if the file data does not exceed 2.1 GB in length.

- IDL 5.5 and later versions will retain the ability to read the 64-bit offset files produced by IDL 5.4.x, thus ensuring backwards compatibility.

- SAVE files written by IDL 5.5 or later versions that contain file data exceeding 2.1GB in length are not readable by older versions of IDL, but will be readable by IDL 5.5 and later versions of IDL that have !VERSION.MEMORY_BITS equal to 64.

- The CONVERT_SR54 procedure, a part of the IDL 5.5 user library, can be used to convert SAVE files written within IDL 5.4 into the newer IDL 5.5

format. This allows existing data files to become readable by previous IDL versions. The CONVERT_SR54 procedure is located in the *IDL_DIR*/lib/obsolete directory.

# Chapter 5
# Creating Procedures and Functions

The following topics are covered in this chapter:

# Overview of Procedures and Functions

Procedures and functions are self-contained modules that break large tasks into smaller, more manageable ones. Modular programs simplify debugging and maintenance and, because they are reusable, minimize the amount of new code required for each application.

New procedures and functions can be written in IDL and called in the same manner as the system-defined procedures or functions from the command prompt or from other programs. When a procedure or function is finished, it executes a RETURN statement that returns control to its caller. Functions always return an explicit result.

A procedure is called by a procedure call statement, while a function is called by a function reference. For example, if myproABC is a procedure and myfuncXYZ is a function, the calling syntax is:

```
; Call procedure with two parameters.
myproABC, A, 12

; Call function with one parameter. The result is stored
; in variable A.
A = myfuncXYZ(C/D)
```

**Note** ──────────────────────────────────────────────────────────

See Chapter 6, "Library Authoring" for information on naming procedures to avoid conflicts with IDL routine names. It is important to implement and consistently use a naming scheme from the earliest stages of code development.

──────────────────────────────────────────────────────────────────

Procedures and functions are collectively referred to as *routines*. An IDL program file may contain one or many routines, which can be a mix of procedures and functions.

# Defining a Procedure

A sequence of one or more IDL statements can be given a name, compiled, and saved for future use with the procedure definition statement. Once a procedure has been successfully compiled, it can be executed using a procedure call statement interactively from the terminal, from a main program, or from another procedure or function.

The general format for the definition of a procedure is as follows:

```
PRO Name, Parameter1, ..., Parametern
   ; Statements defining procedure.
   Statement1
   Statement2
   ...
; End of procedure definition.
END
```

The PRO statement must be the first line in a user-written IDL procedure.

Calling a user-written procedure that is in a directory in the IDL search path (!PATH) and has the same name as the prefix of the .sav or .pro file, causes the procedure to be read from the disk, compiled, and executed without interrupting program execution.

## Calling a Procedure

The syntax of the procedure call statement is as follows:

```
Procedure_Name, Parameter1, Parameter2, ..., Parametern
```

The procedure call statement invokes a system, user-written, or externally-defined procedure. The parameters that follow the procedure's name are passed to the procedure. When the called procedure finishes, control resumes at the statement following the procedure call statement. Procedure names can be up to 128 characters long.

Procedures can come from the following sources:

- System procedures provided with IDL.

- User-written procedures written in IDL and compiled with the .RUN command.

- User-written procedures that are compiled automatically because they reside in directories in the search path. These procedures are compiled the first time they are used. See

- Procedures written in IDL, that are included with the IDL distribution, located in directories that are specified in the search path.

- Under many operating systems, user-written system procedures coded in FORTRAN, C, or any language that follows the standard calling conventions, which have been dynamically linked with IDL using the LINKIMAGE or CALL_EXTERNAL procedures.

# Procedure Examples

Some procedures can be called without any parameters. For example:

```
IPLOT
```

This is a procedure call to launch the iPlot iTool. There are no explicit inputs or outputs. You can also call iPlot with parameters including data and color specifications:

```
data = RANDOMU(Seed,45)
IPLOT, data, COLOR=[255,0,0]
```

This opens the iPlot tool and passes it random plot data. The data parameter is an argument and the COLOR parameter is a keyword. These elements are described in more detail in "Parameters" on page 82.

You can also create a named program consisting of a procedure. For example, suppose you have a file called hello_world.pro containing the following code:

```
PRO hello_world
   PRINT, 'Hello World'
END
```

This IDL "program" consists of a single *user-defined procedure*.

IDL program files are assumed to have the extension .pro or the extension .sav. When IDL searches for a user-defined procedure or function, it searches for files consisting of the name of the procedure or function, followed by the .pro or .sav extension. Procedures and functions can also accept arguments and keywords. Both arguments and keywords allow the program that calls the routine to pass data in the form of IDL variables or expressions to the routine.

For example, the previous user-defined procedure could be changed to include an argument and a keyword:

```
PRO hello_world, name, INCLUDE_NAME = include
   IF (KEYWORD_SET(include) && (N_ELEMENTS(name) NE 0)) THEN BEGIN
      PRINT, 'Hello World From '+ name
   ENDIF ELSE PRINT, 'Hello World'
END
```

Now if the INCLUDE_NAME keyword is set to a value greater than zero, the above procedure will include the string contained within the `name` variable if a value was supplied for the *name* argument. Enter the following procedure call at the command line:

```
hello_world, name, /INCLUDE_NAME
```

IDL prints,

```
Hello World
```

Now define a string name and repeat the procedure call:

```
name = "Horton"
hello_world, name, /INCLUDE_NAME
```

IDL prints:

```
Hello World From Horton
```

This example uses the KEYWORD_SET and N_ELEMENTS functions in order to handle the possibility of missing information in a procedure or function call. See for more information.

# Defining a Function

A function is a program unit containing one or more IDL statements that returns a value. This unit executes independently of its caller. It has its own local variables and execution environment. Referencing a function causes the program unit to be executed. All functions return a function value which is given as a parameter in the RETURN statement used to exit the function. Function names can be up to 128 characters long.

The general format of a function definition is as follows:

```
FUNCTION Name, Parameter₁, ..., Parameterₙ
    Statement₁
    Statement₂
    ...
    ...
    RETURN, Expression
END
```

## Function Example

To define a function called AVERAGE, which returns the average value of an array, use the following statements:

```
FUNCTION AVERAGE, arr
    RETURN, TOTAL(arr)/N_ELEMENTS(arr)
END
```

Once the function AVERAGE has been defined, it is executed by entering the function name followed by its arguments enclosed in parentheses. Assuming the variable X contains an array, the statement,

```
PRINT, AVERAGE(X^2)
```

squares the array X, passes this result to the AVERAGE function, and prints the result. To return the result in a variable, use a function call as follows:

```
vAvg = AVERAGE(X^2)
```

Parameters passed to functions are identified by their position or by a keyword. See "Using Keyword Parameters" on page 86. If a function has no parameters, you must specify empty parentheses in the function call.

# Automatic Compilation and Execution

When used in interactive mode (in the IDL Workbench or in a command shell session — anywhere you see the `IDL>` prompt) IDL automatically compiles and executes a user-written function or procedure when it is first referenced if:

1.  The source code of the function is in the current working directory or in a directory in the IDL search path defined by the system variable !PATH.

2.  The name of the file containing the function is the same as the function name suffixed by `.pro` or `.sav`. The suffix should be in lowercase letters.

IDL does *not* search for files to compile or restore in runtime mode. When using the IDL Virtual Machine or any other runtime mechanism, all routines required by the application must either be restored when the application is initially loaded, or restored explicitly by the application code.

**Note** ─────────────────────────────────────────────

IDL is case-insensitive. However, for some operating systems, IDL only checks for the lowercase filename based on the name of the procedure or function. We recommend that all filenames be lowercase letters.

─────────────────────────────────────────────────────

**Warning** ──────────────────────────────────────────

User-written functions must be defined before they are referenced, unless they meet the above conditions for automatic compilation, or the function name has been reserved by using the FORWARD_FUNCTION statement described below. This restriction is necessary in order to distinguish between function calls and subscripted variable references.

─────────────────────────────────────────────────────

For more information on how to access routines, see

## A Note on Files with Duplicate Names

If multiple `.pro` or `.sav` files with the same base name exist in the directories specified by the !PATH system variable, IDL will compile or restore the first file it finds, according to the following rules:

1.  IDL begins by searching in the current directory.

2.  If a `.pro` file whose base name matches the routine specified exists in the directory being searched, IDL compiles it, runs the routine, and stops searching.

3.  If a `.sav` file whose base name matches the routine specified exists in the directory being searched, IDL restores it, runs the routine, and stops searching.

4.  IDL proceeds to the first directory specified by the !PATH system variable and begins testing again with step 2. If no match is found, IDL proceeds to check each of the directories specified by the !PATH system variable in the order they appear in !PATH.

5.  If no file whose base name matches the routine specified is found in the directories specified by the !PATH system variable, IDL issues an

    ```
    Attempt to call undefined procedure/function
    ```

    error, and halts execution.

# About Calling and Compiling Functions

Versions of IDL prior to version 5.0 used parentheses to indicate array subscripts. Because function calls use parentheses as well, the IDL compiler is not able to distinguish between arrays and functions by examining the statement syntax.

User-defined functions, with the exception of those contained in directories specified by the IDL system variable !PATH, must be compiled before the first reference to the function is encountered. This is necessary because the IDL compiler is unable to distinguish between a reference to a variable subscripted with parentheses and a call to a presently undefined user function with the same name. For example, in the statement:

```
A = XYZ(5)
```

it is impossible to tell by context alone if XYZ is an array or a function.

**Note**

In versions of IDL prior to version 5.0, parentheses were used to enclose array subscripts. While using parentheses to enclose array subscripts will continue to work as in previous version of IDL, we strongly suggest that you use brackets in all new code. See "Array Subscript Syntax: [ ] vs. ( )" on page 309 for additional details.

When IDL encounters references that may be either a function call or a subscripted variable, it searches the current directory, then the directories specified by !PATH, for files with names that match the unknown function or variable name. If one or more files matching the unknown name exist, IDL compiles them before attempting to evaluate the expression. If no function or variable with the given name exists, IDL displays an error message.

There are several ways to avoid this problem:

- Compile the lowest-level functions (those that call no other functions) first, then higher-level functions, and finally procedures.

- Place the function in a file with the same name as the function, and place that file in one of the directories specified by !PATH.

- Use the FORWARD_FUNCTION definition statement to inform IDL that a given name refers to a function rather than a variable. See "FORWARD_FUNCTION" (*IDL Reference Guide*).

- Manually compile all functions before any reference, or use RESOLVE_ROUTINE or RESOLVE_ALL to compile functions.

# Parameters

The variables and expressions passed to the function or procedure from its caller are *parameters*. *Actual parameters* are those appearing in the procedure call statement or the function reference. In the following,

```
; Call procedure with two parameters.
myproABC, A, 12

; Call function with one parameter. The result is stored
; in variable A.
A = myfuncXYZ(C/D)
```

the actual parameters in the procedure call are the variable A and the constant 12, while the actual parameter in the function call is the value of the expression (C/D).

*Formal parameters* are the variables declared in the procedure or function definition. The same procedure or function can be called using different actual parameters from a number of places in other program units.

The correspondence between the actual parameters of the caller and the formal parameters of the called procedure is established by position or by keyword.

## Positional Parameters (Arguments)

A positional parameter, or plain *argument*, is a parameter without a keyword. Just as its name implies, the position of a positional parameter establishes the correspondence—the *n*-th formal positional parameter is matched with the *n*-th actual positional parameter.

## Keyword Parameters

A keyword parameter, which can be either actual or formal, is an expression or variable name preceded by a keyword and an equal sign ("=") that identifies which parameter is being passed.

When calling a routine with a keyword parameter, you can abbreviate the keyword to its shortest, unambiguous abbreviation.

**Note** ───────────────────────────────────────────────────────

Because keyword names can be abbreviated, keywords within a routine must have unique names that do not contain the name of another keyword from the same routine. For example, an error will result from using a keyword named mykeyword and another named mykeyword1 in the same routine.

─────────────────────────────────────────────────────────────────

Keyword parameters can also be specified by the caller with the syntax /KEYWORD, which is equivalent to setting the keyword parameter to 1 (e.g., KEYWORD = 1). The syntax /KEYWORD is often referred to, in the rest of this documentation, as *setting* the keyword.

For example, a procedure is defined with a keyword parameter named TEST.

```
PRO XYZ, A, B, TEST = T
```

The caller can supply a value for the formal (keyword) parameter T with the following calls:

```
; Supply only the value of T. A and B are undefined inside the
; procedure.
XYZ, TEST = A

; The value of A is copied to formal parameter T (note the
; abbreviation for TEST), Q to A, and R to B.
XYZ, TE = A, Q, R

; Variable Q is copied to formal parameter A. B and T are undefined
; inside the procedure.
XYZ, Q
result = FUNCTION(Arg1, Arg2, KEYWORD = value)
```

**Note**

When supplying keyword parameters for a function, keywords are specified *inside* the parentheses.

# Copying Parameters

When a procedure or function is called, the actual parameters are copied into the formal parameters of the procedure or function and the module is executed.

On exit, via a RETURN statement, the formal parameters are copied back to the actual parameters, providing they were not expressions or constants. Parameters can be inputs to the program unit; they can be outputs in which the values are set or changed by the program unit; or they can be both inputs and outputs.

When a RETURN statement is encountered in the execution of a procedure or function, control is passed back to the caller immediately after the point of the call. In functions, the parameter of the RETURN statement is the result of the function.

# Number of Parameters

A procedure or a function can be called with fewer arguments than were defined in the procedure or function. For example, if a procedure is defined with 10 parameters, the user or another procedure can call the procedure with 0 to 10 parameters.

Parameters that are not used in the actual argument list are set to be undefined upon entering the procedure or function. If values are stored by the called procedure into parameters not present in the calling statement, these values are discarded when the program unit exits. The number of actual parameters in the calling list can be found by using the system function N_PARAMS. Use the N_ELEMENTS function to determine if a variable is defined.

# Determining Variable Scope

The ARG_PRESENT function returns TRUE if its parameter will be passed back to the caller. This function is useful in user-written procedures to determine if a created value remains within the scope of the calling routine. ARG_PRESENT helps the caller avoid expensive computations and prevents heap leaks. For example, assume that a procedure exists which depends upon an argument passed by the caller:

```
PRO pass_it, i
```

If the caller does not specify *i*, the program may not function properly. You can check to make sure that an argument was specified by using the following statement:

```
IF ARG_PRESENT(i) THEN BEGIN
```

# Function Parameters Example

An example of an IDL function to compute the digital gradient of an image is shown in the example below. The digital gradient approximates the two-dimensional gradient of an image and emphasizes the edges.

This simple function consists of three lines corresponding to the three required components of IDL procedures and functions: the procedure or function declaration, the body of the procedure or function, and the terminating end statement.

```
FUNCTION GRAD, image
; Define a function called GRAD. Result is ABS(dz/dx) + ABS(dz/dy).

; Evaluate and return the result.
  RETURN, ABS(image - SHIFT(image, 1, 0)) + $
    ABS(image-SHIFT(image, 0, 1))

; End of function.
```

```
END
```

The function has one parameter called IMAGE. There are no local variables. Local variables are variables active only within a module (i.e., they are not parameters and are not contained in common blocks).

The result of the function is the value of the expression used as an argument to the RETURN statement. Once compiled, the function is called by referring to it in an expression. Two examples are shown below.

```
; Store gradient of B in A.
A = GRAD(B)

; Display gradient of IMAGE.
; Access image data and pass to GRAD function.
; Display the gradient.
file=FILEPATH('endocell.jpg', SUBDIRECTORY=['examples','data'])
READ_JPEG, file, image, /GRAYSCALE
result=GRAD(image)
IIMAGE, result
```

# Using Keyword Parameters

A short example of a function that exchanges two columns of a $4 \times 4$ homogeneous, coordinate-transformation matrix is shown below. The function has one positional parameter, the coordinate-transformation matrix T. The caller can specify one of the keywords XYEXCH, XZEXCH, or YZEXCH to interchange the *xy*, *xz*, or *yz* axes of the matrix. The result of the function is the new coordinate transformation matrix defined below.

```
; Function to swap columns of T. XYEXCH swaps columns 0 and 1,
; XZEXCH swaps 0 and 2, and YZEXCH swaps 1 and 2.
FUNCTION SWAP, T, XYEXCH = xy, XZEXCH = xz, YZEXCH = yz

   ; Swap columns 0 and 1 if keyword XYEXCH is set.
   IF KEYWORD_SET(XY) THEN S=[0,1] $

   ; Check to see if xz is set.
   ELSE IF KEYWORD_SET(XZ) THEN S=[0,2] $

   ; Check to see if yz is set.
   ELSE IF KEYWORD_SET(YZ) THEN S=[1,2] $

   ; If nothing is set, return.
   ELSE RETURN, T

   ; Copy matrix for result.
   R = T

; Exchange two columns using matrix insertion operators and
; subscript ranges.
   R[S[1], 0] = T[S[0], *]
   R[S[0], 0] = T[S[1], *]

   ; Return result.
   RETURN, R

END
```

Typical calls to SWAP are as follows:

```
Q = SWAP(!P.T, /XYEXCH)
Q = SWAP(Q, /XYEX)
Q = SWAP(INVERT(Z), YZ = 1)
Q = SWAP(Z, XYE = I EQ 0, XZE = I EQ 1, YZE = I EQ 2)
```

Note that keyword names can abbreviated to the shortest unambiguous string. The last example sets one of the three keywords according to the value of the variable I.

**Note** ————————————————————————————————————————————

Because keywords can be abbreviated, keywords within routines must have unique names that do not contain the name of another keyword from the same routine. For example, an error will result from using a keyword named `mykeyword` and another named `mykeyword1`.

# Determining if a Keyword is Set

The previous function example (in "Using Keyword Parameters" on page 86) uses the system function KEYWORD_SET to determine if a keyword parameter has been passed and if it is nonzero. This is similar to using the condition:

```
IF N_ELEMENTS(P) NE 0 THEN IF P THEN ... ...
```

to test if keywords that have a true/false value are both present and true. The N_ELEMENTS function returns the number of elements contained in any expression or variable. Scalars always have one element. The N_ELEMENTS function returns zero if its parameter is an undefined variable. The result is always a longword scalar. The following example determines if a variable is defined using N_ELEMENTS. It sets the variable abc to zero if it is undefined; otherwise, the variable is not changed.

```
IF N_ELEMENTS(abc) EQ 0 THEN abc = 0
```

The KEYWORD_SET function returns a 1 (true), if its parameter is defined and nonzero; otherwise, it returns zero (false). For example, assume that a procedure is written which performs and returns the result of a computation. If the keyword PLOT is present and nonzero, the procedure also plots its result as follows:

```
; Procedure definition.
PRO XYZ, result, PLOT = plot

; Compute result.
   ...

; Plot result if keyword parameter is set.
   IF KEYWORD_SET(PLOT) THEN PLOT, result

END
```

A call to this procedure that produces a plot is shown in the following statement.

```
XYZ, R, /PLOT
```

# Supplying Values for Missing Keywords

N_ELEMENTS is frequently used to check for omitted plain and keyword arguments. N_PARAMS cannot be used to check for the number of keyword arguments because it returns only the number of plain arguments. (See "Supplying Values for Missing Arguments" on page 90.) An example of using N_ELEMENTS to check for a keyword parameter is as follows:

```
; Display an image with a given zoom factor.
; If factor is omitted, use 4.
PRO ZOOM, image, FACTOR = factor

; Supply default for missing keyword parameter.
IF N_ELEMENTS(factor) EQ 0 THEN factor = 4
```

**Note** ─────────────────────────────────────────

If you use this method, the variable `factor` is defined has having the value 4, even though no value was supplied by the user. If the ZOOM procedure were called within another routine, the variable `factor` would be defined for that routine and for any other routines also called by the routine that called ZOOM. This can lead to unexpected behavior if you pass arguments from one routine to another.

You can avoid this problem by using different variable names inside the routine than are used in calling the routine. For example, if you wanted to supply a default zoom factor in the example above, but did not want to change the value of `factor`, you could use an approach similar to the following:

```
IF N_ELEMENTS(factor) EQ 0 THEN zoomfactor = 4 $
    ELSE zoomfactor = factor
```

You would then set the zoom factor internally using the `zoomfactor` variable, leaving `factor` itself unchanged.

─────────────────────────────────────────

# Supplying Values for Missing Arguments

The N_PARAMS function returns the number of positional arguments (not keyword arguments) present in a procedure or function call. A frequent use is to call N_PARAMS to determine if all arguments are present and if not, to supply default values for missing parameters. For example:

```
; Print values of XX and YY. If XX is omitted, print
; values of YY versus element number.
PRO XPRINT, XX, YY

   ; Check number of arguments.
   CASE N_PARAMS() OF

      ; Single-argument case.
      1: BEGIN

         ; First argument is y values.
         Y = XX

         ; Create vector of subscript indices.
         X = INDGEN(N_ELEMENTS(Y))

         END

      ; Two-argument case.
      2: BEGIN

         ; Copy parameters to local arguments.
         Y = YY & X = XX

         END

      ; Print error message.
      ELSE: MESSAGE, 'Wrong number of arguments'

   ENDCASE

   ; Remainder of procedure.
   ...

END
```

# Keyword Inheritance

Keyword inheritance allows IDL routines to accept keyword parameters not defined in their function or procedure declaration and pass them on to the routines that they call. Routines are able to accept keywords on behalf of the routines they call without explicitly processing each individual keyword. The resulting code is simple, and requires significantly less maintenance. Keyword inheritance is of particular value when writing:

- *Wrapper* routines, which are variations of a system or user-provided routine. Such wrappers usually augment the behavior of another routine in a small way, largely passing arguments and keywords through without interpretation. Keyword inheritance allows such wrappers to be very simple, and benefit from not having to specify all the details of the underlying routine's interface. Maintenance of the wrapper is also greatly simplified, because the wrapper does not require modification every time the underlying routine changes.

- Methods for an object. In an object hierarchy, each subclass has the option of overriding the methods provided by its superclasses. Often, the subclass method calls the superclass version. Keyword inheritance makes it simple to pass on keywords without having to be explicitly aware of them, and without having to be concerned with filtering out those keywords that are not accepted by the superclass method. In addition to enhancing maintainability, this allows subclassing from a base class without having detailed knowledge of its internal implementation, an important consideration for object oriented programming.

There are two steps required to use keyword inheritance in an IDL routine:

1. The routine must declare that it accepts inherited keywords. This is done by specifying either the _EXTRA or _REF_EXTRA keyword in the formal parameter list of the routine (note the leading underscore in these names). IDL will use one of its two available keyword inheritance mechanisms depending on which of these keyword parameters is used. The first inheritance mechanism (_EXTRA) passes keywords by *value*, while the other (_REF_EXTRA) passes them by *reference*. The difference between these methods is explained in "Keyword Inheritance Mechanisms" on page 92. Advice on how to choose the best one for your needs can be found in "Choosing a Keyword Inheritance Mechanism" on page 94. Only one of these two keywords can be specified for a given routine.

2.   The routine passes the inherited keywords to a called routine, by including either the _EXTRA or _STRICT_EXTRA keyword in the call to that routine. _EXTRA and _STRICT_EXTRA differ only in how IDL behaves when an inherited keyword is not accepted by the called routine. _EXTRA causes such keywords to be quietly ignored, while _STRICT_EXTRA causes IDL to issue an error and stop execution. _EXTRA is the usual choice, while _STRICT_EXTRA is used primarily for wrapper routines.

When using keyword inheritance, the following points should be kept in mind:

- The mechanism used by a routine for inherited keywords is solely determined by which keyword (_EXTRA or _REF_EXTRA) is used in the formal parameter list for that routine. Hence, _REF_EXTRA is only used in the formal parameter list of a routine, and never in a call to that routine. This also means that you can change an existing routine from using one mechanism to the other by simply changing the name of the keyword. There is no need to change any of the calls to the routine, just the formal parameter list of the routine itself.

- Attempting to use both the _EXTRA and _REF_EXTRA keywords together in the formal parameter list of a function or procedure will cause an error to be issued. You can only use one or the other.

- Only the caller of a routine can dictate whether keywords that are not understood by the called routine should be ignored (_EXTRA) or should generate an error (_STRICT_EXTRA). For this reason, _STRICT_EXTRA is only used in a call to a routine, and not in the formal parameter list for the routine.

- Attempting to use both the _EXTRA and _STRICT_EXTRA keywords together in a call to a function or procedure will cause an error to be issued. You can only use one or the other.

# Keyword Inheritance Mechanisms

As described above, there are two possible mechanisms used by IDL to pass inherited keywords. The one used by a routine is determined by the formal parameter list of the routine.

## _EXTRA: Passing Keyword Parameters by Value

You can cause inherited keyword parameters to be passed to a routine by *value* by adding the keyword parameter _EXTRA to the formal argument list of that routine. Passing parameters by value means that you are giving the called routine a *copy* of

the value of the passed parameter, and not the original. As such, any changes made to the value of such a keyword is not passed back to the caller.

When a routine is defined with the formal keyword parameter _EXTRA, and keywords that are not recognized by that routine are passed to it in a call, IDL constructs an anonymous structure to contain the keyword inheritance information. Each tag in this structure has the name of an inherited keyword, and the value of that tag is a copy of the value that was passed to that keyword. If no unrecognized keywords are passed in a call, the value of the _EXTRA keyword will be *undefined*, indicating that no inherited keyword parameters were passed.

### Modifying Inherited Keyword Values

If extra keyword parameters have been passed by value, their values are stored in an anonymous structure. The inheriting routine has the opportunity to modify these values and/or to filter them prior to passing them to another routine. The CREATE_STRUCT, N_TAGS, and TAG_NAMES functions can all be of use in performing such operations. For example, here is an example of adding a keyword named COLOR with value 12 to an _EXTRA structure:

```
PRO SOMEPROC, _EXTRA = ex
   if (N_ELEMENTS(ex) NE 0) $
      THEN ex = CREATE_STRUCT('COLOR', 12, ex) $
      ELSE ex = { COLOR : 12 }
   SOME_UNDERLYING_PROC, _EXTRA=ex
END
```

The use of N_ELEMENTS is necessary because if the caller does not supply any inherited keyword, the variable EX will have an undefined value, and an attempt to use that value with CREATE_STRUCT will cause an error to be issued. Hence, we only use CREATE_STRUCT if we know that inherited keywords are present.

## _REF_EXTRA: Passing Keyword Parameters by Reference

You specify that a routine accepts inherited keywords by reference, by adding the keyword _REF_EXTRA to the formal argument list of the routine. When a routine is defined with _REF_EXTRA, inherited keywords are passed using IDL's standard parameter passing mechanism, as with any other variable. Unlike regular variables however, the values of these keywords are not available within the routine itself. Instead, the names of these keywords are passed as a string array to the routine as the value of the _REF_EXTRA keyword. The presence of a name in the _REF_EXTRA value indicates that a keyword of that name was passed, and its value is available to be passed on in a function or procedure call (using either _EXTRA or _STRICT_EXTRA). If no unrecognized keywords are passed in a call, the value of

the _EXTRA keyword will be *undefined*, indicating that no inherited keyword parameters were passed.

If inherited keywords passed by reference are modified by a called routine, those changes will be passed back to the caller.

The *pass by reference* keyword inheritance mechanism is especially useful when writing object methods.

### Selective Keyword Redirection

If extra keyword parameters have been passed by reference, you can direct different inherited keywords to different routines by specifying a string or array of strings containing keyword names via the _EXTRA keyword. For example, suppose that we write a procedure named SOMEPROC that passes extra keywords by reference:

```
PRO SOMEPROC, _REF_EXTRA = ex
   ONE, _EXTRA=['MOOSE', 'SQUIRREL']
   TWO, _EXTRA='SQUIRREL'
END
```

If we call the SOMEPROC routine with three keywords:

```
SOMEPROC, MOOSE=moose, SQUIRREL=3, SPY=PTR_NEW(moose)
```

- it will pass the keywords MOOSE and SQUIRREL and their values (the IDL variable moose and the integer 3, respectively) to procedure ONE,

- it will pass the keyword SQUIRREL and its value to procedure TWO,

- it will do nothing with the keyword SPY, or any other keyword that might be passed to it.

# Choosing a Keyword Inheritance Mechanism

The two available keyword inheritance mechanisms have different strengths and weaknesses. The one to choose depends on the requirements of your routine:

- If your routine needs to see the values of the inherited keywords, and you do not need to pass modified values back to the caller, use _EXTRA (pass by value).

- If your routine does not need to see the values of the inherited keywords, and it is OK to pass back modified keyword values, use _REF_EXTRA (pass by reference).

- If your routine is an object method, _REF_EXTRA is most likely the correct choice for your application.

- If either mechanism will serve your needs, as is often the case, then we recommend _REF_EXTRA, which has a minor efficiency advantage over _EXTRA, due to the fact that it does not have to construct an anonymous structure and copy the original values into it.

# Example: Writing a Wrapper Routine

One of the most common uses for the keyword inheritance mechanism is to create *wrapper* routines that extend the functionality of existing routines. This example shows how to write such a wrapper, using both available inheritance mechanisms.

## By Value

In most wrapper routines, there is no need to return modified keyword values back to the calling routine — the aim is simply to provide the complete set of keywords available to the existing routine from the wrapper routine. Hence, the by value form (_EXTRA) of keyword inheritance can be used.

For example, suppose that procedure TEST is a wrapper to the PLOT procedure. The text of such a procedure is shown below:

```
PRO TEST, a, b, _EXTRA = e, COLOR = color
    PLOT, a, b, COLOR = color, _EXTRA = e
END
```

This wrapper passes all keywords it does not accept directly to PLOT using keyword inheritance. If such a keyword is not accepted by the PLOT procedure, it is quietly ignored. If you wish to catch such errors, you would re-write TEST to use the _STRICT_EXTRA keyword in the call to PLOT:

```
PRO TEST, a, b, _EXTRA = e, COLOR = color
    PLOT, a, b, COLOR = color, _STRICT_EXTRA = e
END
```

This definition of the TEST procedure causes unrecognized keywords (any keywords other than COLOR) to be placed into an anonymous structure assigned to the variable *e*. If there are no unrecognized keywords, *e* will be undefined.

For example, when procedure TEST is called with the following command:

```
TEST, x, y, COLOR=3, LINESTYLE = 4, THICK=5
```

variable *e*, within TEST, contains an anonymous structure with the value:

```
{ LINESTYLE: 4, THICK: 5 }
```

These keyword/value pairs are then passed from TEST to the PLOT routine using the _EXTRA keyword:

```
PLOT, a, b, COLOR = color, _EXTRA = e
```

Note that keywords passed into a routine via _EXTRA override previous settings of that keyword. For example, the call:

```
PLOT, a, b, COLOR = color, _EXTRA = {COLOR: 12}
```

specifies a color index of 12 to PLOT.

## By Reference

It is extremely simple to modify the by value (_EXTRA) version of the TEST procedure from the previous section to use by reference keyword inheritance. It suffices to change the _EXTRA keyword to _REF_EXTRA in the formal parameter list:

```
PRO TEST, a, b, _REF_EXTRA = e, COLOR = color
    PLOT, a, b, COLOR = color, _STRICT_EXTRA = e
END
```

This definition of the TEST procedure causes unrecognized keywords (any keywords other than COLOR) to be passed to PLOT using the normal IDL parameter passing mechanism. However, their values are not visible within TEST itself. Instead, a string array containing the inherited keyword names is assigned to the variable $e$. If there are no unrecognized keywords, e will be undefined.

For example, when procedure TEST is called with the following command:

```
TEST, x, y, COLOR=3, LINESTYLE = 4, THICK=5
```

variable *e*, within TEST, contains an anonymous structure with the value:

```
[ 'LINESTYLE', 'THICK' ]
```

These inherited keywords are then passed from TEST to the PLOT routine using the _EXTRA keyword. Note that keywords passed into a routine via _EXTRA override previous settings of that keyword. For example, the call:

```
PLOT, a, b, COLOR = color, _EXTRA = {COLOR: 12}
```

specifies a color index of 12 to PLOT. Also note that we are passing a structure (the by value format used by _EXTRA) as the value of the extra keyword to a routine that uses the by reference keyword inheritance mechanism (_REF_EXTRA). There is no problem in doing this, because each routine establishes its own inheritance mechanism independent of any other routines that may be calling it. However, any keyword values that are changed within PLOT will fail to be returned to the caller due to the use of the by-value mechanism.

## Example: By Value Versus By Reference

The *pass by reference* keyword inheritance mechanism allows you to change the value of a variable in the calling routine's context from within the routine, whereas the *pass by value* mechanism does not. To demonstrate this difference between _EXTRA and _REF_EXTRA, consider the following simple example procedures:

```
PRO HELP_BYVAL, _EXTRA = ex
  HELP, _EXTRA = ex
END

PRO HELP_BYREF, _REF_EXTRA = ex
  HELP, _EXTRA = ex
END
```

Both HELP_BYVAL and HELP_BYREF are simple wrappers to the HELP procedure. The HELP procedure accepts a keyword named OUTPUT that passes back a value to the caller. Observe the result when we call each wrapper, specifying OUTPUT as an inherited keyword parameter:

```
HELP_BYVAL, OUTPUT = out & HELP, out
```

IDL prints:

```
% At HELP_BYVAL        2 /dev/tty
%    $MAIN$
EX              UNDEFINED = <Undefined>
Compiled Procedures:
    $MAIN$  HELP_BYVAL

Compiled Functions:

OUT             UNDEFINED = <Undefined>
```

This occurs because the HELP call within HELP_BYVAL is passed a variable that cannot be used to return a value, due to the use of by value keyword inheritance. It therefore reverts to the default of writing to the user's screen, and no value is returned to the caller for the OUTPUT keyword.

Now run HELP_BYREF:

```
HELP_BYREF, OUTPUT = out & HELP, out
```

IDL prints:

```
OUT             STRING    = Array[8]
```

HELP_BYREF returns the value of the HELP OUTPUT keyword as desired.

# Entering Procedure Definitions

Procedures and functions are compiled using the .RUN or .COMPILE executive commands. The format of these commands is as follows:

```
.RUN [File_1 , File_n, ... ]
.COMPILE [File_1 , File_n, ... ]
```

From 1 to 10 files, each containing one or more program units, can be compiled. For more information, see ".RUN" and ".COMPILE" (*IDL Reference Guide*).

To enter program text directly from the keyboard, simply enter .RUN at the IDL> prompt. IDL will prompt with the "-" character, indicating that it is compiling a directly entered program. As long as IDL requires more text to complete a program unit, it prompts with the "-"character. Rather than executing statements immediately after they are entered, IDL compiles the program unit as a whole. See "Creating $MAIN$ Programs" on page 22 for more information.

Procedure and function definition statements cannot be entered in the single-statement mode, but must be prefaced by either .RUN or .RNEW.

The first non-empty line the IDL compiler reads determines the type of the program unit: procedure, function, or main program. If the first non-empty line is not a procedure or function definition statement, the program unit is assumed to be a main program. The name of the procedure or function is given by the identifier following the keyword PRO or FUNCTION. If a program unit with the same name is already compiled, it is replaced by the new program unit.

# How IDL Resolves Routines

When IDL encounters a call to a function or procedure, it must find the routine to call. To do this, it goes through the following steps. If a given step yields a callable routine, IDL arranges to call that routine and the search ends at that point:

1. If the routine is known to be a built-in intrinsic routine (commonly referred to as a *system routine*), then IDL calls that system routine.

2. If a user routine written in the IDL language with the desired name has already been compiled, IDL calls that routine.

3. If a file with the name of the desired routine (and ending with the filename suffix `.pro`) exists in the current working directory, IDL assumes that this file contains the desired routine. It arranges to call a user routine, but does not compile the file. The file will be compiled when IDL actually needs it. In other words, it is compiled at run time when IDL actually attempts to call the routine, not when the code for the call is compiled.

4. IDL searches the directories given by the !PATH system variable for a file with the name of the desired routine ending with the filename suffix `.pro`. If such a file exists, IDL assumes that this file contains the desired routine. It arranges to call a user routine, but does not compile the file, as described in the previous step.

5. If the above steps do not yield a callable routine, IDL either assumes that the name is an array (due to the ambiguity inherent in allowing parentheses to indicate either functions or arrays) or that the desired routine does not exist (See Chapter 15, "Arrays" for a discussion of this ambiguity). In either case, the result is not a callable routine.

# Parameter Passing Mechanism

Parameters are passed to IDL system and user-written procedures and functions by *value* or by *reference*. It is important to recognize the distinction between these two methods.

- Expressions, constants, system variables, and subscripted variable references are passed by value.

- Variables are passed by reference.

Parameters passed by value can only be inputs to program units. Results cannot be passed back to the caller by these parameters. Parameters passed by reference can convey information in either or both directions. For example, consider the following trivial procedure:

```
PRO ADD, A, B
   A = A + B
   RETURN
END
```

This procedure adds its second parameter to the first, returning the result in the first. The call

```
ADD, A, 4
```

adds 4 to A and stores the result in variable A. The first parameter is passed by reference and the second parameter, a constant, is passed by value.

The following call does nothing because a value cannot be stored in the constant 4, which was passed by value.

```
ADD, 4, A
```

No error message is issued. Similarly, if ARR is an array, the call

```
ADD, ARR[5], 4
```

will not achieve the desired effect (adding 4 to element ARR[5]), because subscripted variables are passed by value. The correct, though somewhat awkward, method is as follows:

```
TEMP = ARR[5]
ADD, TEMP, 4
ARR[5] = TEMP
```

**Note** ───────────────────────────────────────────

IDL structures behave in two distinct ways. Entire structures are passed by reference, but individual structure fields are passed by value. See "Parameter Passing with Structures" on page 345 for additional details.

# Calling Mechanism

When a user-written procedure or function is called, the following actions occur:

1. All of the actual arguments in the user-procedure call list are evaluated and saved in temporary locations.

2. The actual parameters that were saved are substituted for the formal parameters given in the definition of the called procedure. All other variables local to the called procedure are set to undefined.

3. The function or procedure is executed until a RETURN or RETALL statement is encountered. Procedures also can return on an END statement. The result of a user-written function is passed back to the caller by specifying it as the value of a RETURN statement. RETURN statements in procedures cannot specify a return value.

4. All local variables in the procedure, those variables that are neither parameters nor common variables, are deleted.

5. The new values of the parameters that were passed by reference are copied back into the corresponding variables. Actual parameters that were passed by value are deleted.

6. Control resumes in the calling procedure after the procedure call statement or function reference.

## Recursion

Recursion (i.e., a program calling itself) is supported for both procedures and functions.

## Example

Here is an example of an IDL procedure that reads and plots the next vector from a file. This example illustrates using common variables to store values between calls, as local parameters are destroyed on exit. It assumes that the file containing the data is open on logical unit 1 and that the file contains a number of 512-element, floating-point vectors.

```
; Read and plot the next record from file 1. If RECNO is specified,
; set the current record to its value and plot it.
PRO NXT, recno

   ; Save previous record number.
```

```
COMMON NXT_COM, lastrec

; Set record number if parameter is present.
IF N_PARAMS(0) GE 1 THEN lastrec = recno

; Define LASTREC if this is first call.
IF N_ELEMENTS(lastrec) LE 0 THEN lastrec = 0

; Define file structure.
AA = ASSOC(1, FLTARR(512))

; Read and plot record.
PLOT, AA[lastrec]

; Increment record for next time.
lastrec = lastrec + 1

END
```

Once the user has opened the file, typing NXT will read and plot the next record. Typing NXT, N will read and plot record number N.

# Calling Functions/Procedures Indirectly

The CALL_FUNCTION and CALL_PROCEDURE routines are used to indirectly call functions and procedures whose names are contained in strings. The CALL_METHOD routine can be used to indirectly call an object method whose name is contained in a string. Although not as flexible as the EXECUTE function (see "EXECUTE" (*IDL Reference Guide*)), the CALL_* routines are much faster, and should be used in preference to EXECUTE whenever possible.

## Example

This example code fragment, taken from the routine SVDFIT, calls a function whose name is passed to SVDFIT via a keyword parameter as a string. If the keyword parameter is omitted, the function POLY is called.

```
; Function declaration.
FUNCTION SVDFIT,..., FUNCT = funct

   ...

   ; Use default name, POLY, for function if not specified.
   IF N_ELEMENTS(FUNCT) EQ 0 THEN FUNCT = 'POLY'

   ; Make a string of the form "a = funct(x,m)", and execute it.
   Z = EXECUTE('A = '+FUNCT+'(X,M)')

   ...
```

The above example is easily made more efficient by replacing the call to EXECUTE with the following line:

```
A = CALL_FUNCTION(FUNCT, X, M)
```

# Chapter 6
# Library Authoring

The following topics are covered in this chapter:

# Overview of Library Authoring

Library authors provide an invaluable resource to the IDL community — they develop domain-specific programs and applications that implement knowledge far beyond our level of expertise. User library code is often freely available, supported, and documented. However, as the number of library authors and routines continues to grow, it becomes increasingly important for authors to adhere to a routine naming convention within their libraries that avoids conflicts with core IDL functionality.

Most user libraries start out as small collections of code, and then grow. Initially, the naming issue is not very important. Over time, the library grows in complexity and number of users. Because this is often a gradual process, the importance of naming is not obvious until there is a conflict with IDL system functionality, or a conflict with another library author's code.

An understanding of the way IDL resolves routines during program execution reveals why new IDL system procedures and functions may periodically conflict with pre-existing routines written by users in the IDL community. (See "How IDL Resolves Routines" on page 99 for step-by-step routine resolution details.)

The fact that IDL system routines always take precedence over user routines provides the following benefits:

- The IDL environment remains reliable and consistent — a call to FFT always returns the IDL version of the FFT function.

- It eliminates a great deal of path searching, which translates into faster execution speed.

In contrast, if user routines took precedence over system routines, a given installation could radically alter the meaning of common and basic IDL constructs simply by creating user routines with the names of IDL system routines. This would result in conflicts when sharing code, degradation of the common IDL language core, and ultimately, the reduced usefulness of IDL.

Although the way IDL handles the search for routines is simple, efficient, and reliable, it is not perfect. The potential for namespace conflicts exists. It is important to recognize and take steps to avoid these naming conflicts as described in the following sections:

- "Recognizing Potential Naming Conflicts" on page 107
- "Advice for Library Authors" on page 110
- "Converting Existing Libraries" on page 111

# Recognizing Potential Naming Conflicts

IDL favors simple names, and it blurs the user level distinction between system routines and user routines. The reason for this has everything to do with IDL's orientation towards *ad hoc* analysis. The primary goal is transparency. Names should make sense, be easy to remember, and not require too much typing. Language transparency also results in very human-readable code. In conjunction with the way IDL searches for routines, this may cause either user level or system level conflicts.

## User Level Conflicts

In the user level case, an IDL user writes a routine that is not part of the base release of IDL, and places it in a local library. At some later date, a new version of IDL is installed that contains a new IDL library routine with the same name as the user's routine. Depending on the order of the directories in the user's path, one of these two routines is executed. If the user's routine is used, IDL library code that calls the routine will get the wrong version and fail in strange and mysterious ways. If the IDL routine is used, the IDL library will be satisfied, but the user's library will get the wrong version, also with bad results.

## System Level Conflicts

The system level case is similar, but harder to work around. In this case, the user creates a local routine, as before. However, the new version of IDL contains a system routine with the same name. In this case, IDL will always choose to use the system routine, and the user routine simply vanishes from view never to be called again. The order of the search path is meaningless in this case because the search path is not even consulted. A system routine always has precedence over a user routine.

# Choosing Routine Names to Avoid Conflicts

Naming conflicts can result in costly and time consuming problems; carefully considered names make everything easier. On the surface, naming routines seems like a trivial issue, but names are very important. It is crucial to adopt and consistently adhere to a routine naming strategy to avoid conflict. The core idea of this convention (described in detail in "Advice for Library Authors" on page 110) is to prefix all library routine names with a unique identifier, one indicative of your organization or project. We reserve routine names that are generic, and those with an "IDL" prefix on behalf of the entire IDL community. Prefixing your user library routines significantly reduces the risk of namespace collisions with IDL routines.

As a library author, your decision to follow a routine prefixing strategy benefits the entire IDL community. This convention translates into simplicity and reliability, allowing IDL system routines to always take precedence over user routines. It also raises the visibility of your routines, readily distinguishing them as part of your library.

**Note** ───────────────────────────────────────────────────────────
For instructions on how to prefix an existing user library, see "Converting Existing Libraries" on page 111.
─────────────────────────────────────────────────────────────────

# Cross-Platform Naming of IDL .pro Files

When naming IDL .pro files used in cross-platform applications, be aware of the various platforms' file naming conventions and limitations. For example, the ":" character is not allowed in a filename under Microsoft Windows.

Be careful with case when naming files. For example, while Microsoft Windows systems present file names using mixed case, file names are in fact case-insensitive. Under Unix, file names are case sensitive—file.pro is different from File.pro.

When writing cross-platform applications, you should avoid using filenames that are different only in case. The safest course is to use filenames that are all lower case.

Remember, too, that IDL commands are themselves case-insensitive. If entered at the IDL command prompt, the following are equivalent:

```
IDL> command
IDL> COMMAND
IDL> CommanD
```

## Automatic Compilation and Case Sensitivity

On UNIX platforms, where filename case matters, IDL looks for a lower-case filename when you enter the name of a user-written routine at the IDL command prompt. Thus, if you save your program file as myprogram.pro and enter the following at the IDL command prompt:

```
IDL> MyProgram
```

IDL will compile the file myprogram.pro and attempt to execute a procedure named myprogram.

If you save your program file as MyProgram.pro and enter the following at the IDL command prompt:

```
IDL> MyProgram
```

IDL will *not* compile the file MyProgram.pro and will issue an error that looks like:

```
% Attempt to call undefined procedure/function: 'MYPROGRAM'.
% Execution halted at: $MAIN$
```

You can compile and run a program with a mixed- or upper-case file name on a UNIX platform by using IDL's .COMPILE or .RUN executive commands:

```
IDL> .COMPILE MyProgram
IDL> MyProgram
```

or, if MyProgram.pro contains a main-level program:

```
IDL> .RUN MyProgram
```

In general we recommend that you use lower-case file names on platforms where case matters.

# Advice for Library Authors

An ordinary IDL programmer needs only to solve his or her own problems to the desired level of quality, reusability, and robustness. Life is more difficult for an author of a library of IDL routines. In addition to the challenges facing any programmer, library authors face additional challenges:

- The structure and organization of the library needs to encourage reuse and generality.

- Library code must be more robust than the usual program. Stability of implementation, and especially of interface, are very important.

- Errors must be gracefully handled whenever possible. See Chapter 8, "Debugging and Error-Handling" for more on error control.

- The most useful libraries are written to work correctly on a wide variety of platforms, without requiring their users to be aware of the details.

- Documentation must be provided, or the library will not find users.

- Libraries must be able to co-exist with other code over which they have no control. Authors must not alter the global environment in ways that cause conflicts, and they must also take care to prefix the names of all routines, common blocks, systems variables, and any other global resources they use. This prevents a library from conflicting with other libraries on the same system, and protects the library from changes to IDL that may occur in newer releases.

## Prefixing Routine Names

The use of a proper prefix minimizes the risk of a namespace collision as described in "Recognizing Potential Naming Conflicts" on page 107. In selecting a prefix for your library, you should select a name that is short, mnemonic, and unlikely to be chosen by others. For example, such a name might use the name of your organization or project in an abbreviated form.

Non-prefixed names and names prefixed by "IDL" are reserved. New names of these forms can and will appear without warning in new versions of IDL, and should be avoided when naming new library routines.

# Converting Existing Libraries

Many libraries that already exist do not follow the naming guidelines provided in "Advice for Library Authors" on page 110. Such libraries are bound to experience an occasional conflict with new versions of IDL. The best solution to avoid conflicts is to perform a systematic one-time conversion to a prefixed naming scheme.

Any existing library is likely to already have users. Assuming that non-prefixed names were used in such libraries, it is not possible to simply change the names. Such conversions require time to carry out, and once that has happened, it takes time for users to adjust and alter their usage. However, the actual conversion can go very quickly, and with proper planning it is easy to offer a backwards compatibility option for your users. Use the following steps to convert an existing library:

1.  Generate a list of all files containing routines to be renamed.

2.  Using this list, build an IDL batch file that uses .COMPILE on each file.

3.  Start a fresh IDL session, execute the batch file, and use HELP, /ROUTINES to get a complete list of all compiled routines. Only IDL user library routines (those `.pro` files shipped with the IDL distribution) should not contain a prefix.

4.  As you rename each routine to its prefixed form, write a non-prefixed wrapper routine with the old name that calls the new version. Such wrappers are easy to write in IDL, using the _REF_EXTRA keyword to pass keywords through to the real routine. See "Keyword Inheritance" on page 91 for details.

5.  Use the COMPILE_OPT OBSOLETE compilation directive in such wrappers so that IDL will recognize them as obsolete routines. See COMPILE_OPT in the *IDL Reference Guide* for more information on COMPILE_OPT. These compatibility wrappers serve the following purposes:

    •   You can use them to migrate your library to fully prefixed form over time, since the wrapper will be used any place you failed to change to calling the new name. This enhances the stability of the library and gives you time to do a careful job.

    •   Once you are finished, you can provide them to your customers as a bridge, so that their old code continues to work.

    •   As you change the names of routines, use grep (or a similar file searching tool) to locate uses of that name, and convert them to the new form as well.

6.  Iterate, using the batch file mentioned above to find any remaining non-prefixed uses of the library names. Since your wrappers specified the

COMPILE_OPT OBSOLETE directive, you can set the !WARN system variable to help you pinpoint such uses. You are done when your batch file reveals no more unprefixed names.

Once the conversion is done, you can use the compatibility wrappers to smoothly transition your users to the new names. You should keep the wrappers in a separate subdirectory, and even consider making them optional. Doing this raises the end user's awareness of the issue and may convince them to convert to using the new names sooner rather than later.

When you add new routines to your library, ensure that they use the proper prefix. Do not provide non-prefixed wrapper routines for new routines. There is no backward compatibility issue in this case, and they are not needed.

Although the one time hit of prefixing an existing library can consume some time and effort, there are benefits that accrue from doing it. When new versions of IDL are released, the odds of the library working with the new version without encountering any name clashes are extremely high. Use of a consistent prefix also raises the profile of the library to the end user, raising their level of understanding and appreciation for the work it does.

# Chapter 7
# Program Control

The following topics are covered in this chapter:

# Overview of Program Control

IDL contains various *program control* constructs for controlling the flow of program execution, such as conditional expressions and looping mechanisms. These constructs include the following.

## Compound Statements

Use BEGIN and END to create a block of statements, which is simply a group of statements that are the subject of a conditional or repetitive statement.

- BEGIN...END

## Conditional Statements

Most useful applications have the ability to perform different actions in response to different conditions. This decision-making ability is provided in the form of *conditional statements*.

- IF...THEN...ELSE
- CASE
- SWITCH

## Loop Statements

Loop statements perform the same set of statements multiple times. Rather than repeat a set of statements again and again, a loop can be used to perform the same set of statements repeatedly.

- FOR...DO
- REPEAT...UNTIL
- WHILE...DO

**Note** ──────────────────────────────────────────

IDL's array capabilities can often be used in place of loops to write much more efficient programs. For example, if you want to perform the same calculation on each element of an array, you could write a loop to iterate over each array element:

```
array = INDGEN(10)
FOR i = 0,9 DO BEGIN
   array[i] = array[i] * 2
ENDFOR
```

This is much less efficient than using IDL's built-in array capabilities:

```
array = INDGEN(10)
array = array * 2
```

See "Use Vector and Array Operations" on page 196 for details.

─────────────────────────────────────────────────

# Jump Statements

Jump statements can modify the behavior of conditional and iterative statements.

- BREAK
- CONTINUE
- GOTO

# Compound Statements

Many of the language constructs that we will discuss in this chapter evaluate an expression, then perform an action based on whether the expression is true or false, such as with the IF statement:

```
IF expression THEN statement
```

For example, we would say "If X equals 1, then set Y equal to 2" as follows:

```
IF (X EQ 1) THEN Y = 2
```

But what if we want to do more than one thing if X equals 1? For example, "If X equals 1, set Y equal to 2 and print the value of Y." If we wrote it as follows, then the PRINT statement would always be executed, not just when X equals 1:

```
IF (X EQ 1) THEN Y = 2
PRINT, Y
```

IDL provides a container into which you can put multiple statements that are the subject of a conditional or repetitive statement. This container is called a BEGIN...END block, or *compound statement*. A compound statement is treated as a single statement and can be used anywhere a single statement can appear.

## BEGIN...END

The BEGIN...END statement is used to create a block of statements, which is simply a group of statements that are treated as a single statement. Blocks are necessary when more than one statement is the subject of a conditional or repetitive statement.

For example, the above code could be written as follows:

```
IF (X EQ 1) THEN BEGIN
    Y = 2
    PRINT, Y
END
```

All the statements between the BEGIN and the END are the subject of the IF statement. The group of statements is executed as a single statement. Syntactically, a block of statements is composed of one or more statements of any type, started by BEGIN and ended by an END identifier. To be syntactically correct, we should have ended our block with ENDIF rather than just END:

```
IF (X EQ 1) THEN BEGIN
    Y = 2
    PRINT, Y
ENDIF
```

This is to ensure proper nesting of blocks. The END identifier used to terminate the block should correspond to the type of statement in which BEGIN is used. The following table lists the correct END identifiers to use with each type of statement.

| Statement | END Identifier | Example |
|---|---|---|
| ELSE BEGIN | ENDELSE | ```IF (0) THEN A=1 ELSE BEGIN`<br>`    A=2`<br>`ENDELSE``` |
| FOR *variable=init*, *limit* DO BEGIN | ENDFOR | ```FOR i=1,5 DO BEGIN`<br>`    PRINT, array[i]`<br>`ENDFOR``` |
| IF *expression* THEN BEGIN | ENDIF | ```IF (0) THEN BEGIN`<br>`    A=1`<br>`ENDIF``` |
| REPEAT BEGIN | ENDREP | ```REPEAT BEGIN`<br>`    A = A * 2`<br>`ENDREP UNTIL A GT B``` |
| WHILE *expression* DO BEGIN | ENDWHILE | ```WHILE ~ EOF(1) DO BEGIN`<br>`    READF, 1, A, B, C`<br>`ENDWHILE``` |
| *LABEL*: BEGIN | END | ```LABEL1: BEGIN`<br>`    PRINT, A`<br>`END``` |
| *case_expression*: BEGIN | END | ```CASE name OF`<br>`'Moe': BEGIN`<br>`        PRINT, 'Stooge'`<br>`      END`<br>`ENDCASE``` |
| *switch_expression*: BEGIN | END | ```SWITCH name OF`<br>`'Moe': BEGIN`<br>`        PRINT, 'Stooge'`<br>`      END`<br>`ENDSWITCH``` |

*Table 7-1: Types of END Identifiers*

**Note**

CASE and SWITCH also have their own END identifiers. CASE should always be ended with ENDCASE, and SWITCH should always be ended with ENDSWITCH.

The IDL compiler checks the end of each block, comparing it with the type of the enclosing statement. Any block can be terminated by the generic END, but no type checking is performed. Using the correct type of END identifier for each block makes it easier to find blocks that you have not properly terminated.

Listings produced by the IDL compiler indent each block four spaces to the right of the previous level to make the program structure easier to read. (See ".RUN" (*IDL Reference Guide*) for details on producing program listings with the IDL compiler.)

# IF...THEN...ELSE

The IF statement is used to conditionally execute a statement or a block of statements. The syntax of the IF statement is as follows:

```
IF expression THEN statement [ ELSE statement ]
```

or

```
IF expression THEN BEGIN
    statements
ENDIF [ ELSE BEGIN
    statements
ENDELSE ]
```

The expression after the "IF" is called the *condition* of the IF statement. This expression (or condition) is evaluated, and if true, the statement following the "THEN" is executed. (See "Definition of True and False" on page 138 for details on how the "truth" of an expression is determined.)

For example:

```
A = 2
IF A EQ 2 THEN PRINT, 'A is two'
```

Here, IDL prints "A is two".

If the expression evaluates to a *false* value, the statement following the "ELSE" clause is executed:

```
A = 3
IF A EQ 2 THEN PRINT, 'A is two' ELSE PRINT, 'A is not two'
```

Here, IDL prints "A is not two".

Control passes immediately to the next statement if the condition is false and the ELSE clause is not present.

**Note**

Another way to write an IF...THEN...ELSE statement is with a conditional expression using the ?: operator. For more information, see "Working with Conditional Expressions" on page 240.

**Tip**

Programs with vector and array expressions run faster than programs with scalars, loops, and IF statements. See "Use Vector and Array Operations" on page 196 for a discussion on increasing efficiency of these expressions.

# Using Statement Blocks with the IF Statement

The THEN and ELSE clauses can be in the form of a block (or group of statements) with the delimiters BEGIN and END (see "BEGIN...END" on page 116). To ensure proper nesting of blocks, you can use ENDIF and ENDELSE to terminate the block, instead of using the generic END. Below is an example of the use of blocks within an IF statement.

```
IF (I NE 0.0) THEN BEGIN
    ...
ENDIF ELSE BEGIN
    ...
ENDELSE
```

# Nesting IF Statements

IF statements can be nested in the following manner:

```
IF P1 THEN S1 ELSE $
IF P2 THEN S2 ELSE $
    ...
IF PN THEN SN ELSE SX
```

If condition P1 is true, only statement S1 is executed; if condition P2 is true, only statement S2 is executed, etc. If none of the conditions are true, statement SX will be executed. Conditions are tested in the order they are written. The construction above is similar to the CASE statement except that the conditions are not necessarily related.

# CASE

The CASE statement is used to select one, and only one, statement for execution, depending upon the value of the expression following the word CASE. This expression is called the case selector expression. The general form of the CASE statement is as follows:

```
CASE expression OF
   expression: statement
   ...
   expression: statement
[ELSE: statement]
ENDCASE
```

Each statement that is part of a CASE statement is preceded by an expression that is compared to the value of the selector expression. CASE executes by comparing the CASE expression with each selector expression in the order written. If a match is found, the statement is executed and control resumes directly below the CASE statement.

The ELSE clause of the CASE statement is optional. If included, it matches any selector expression, causing its code to be executed. For this reason, it is usually written as the last clause in the CASE statement. The ELSE statement is executed only if none of the preceding statement expressions match. If an ELSE clause is not included and none of the values match the selector, an error occurs and program execution stops.

The BREAK statement can be used within CASE statements to force an immediate exit from the CASE.

## Example — Case Statement Use

An example of the CASE statement follows:

```
CASE name OF
   'Larry': PRINT, 'Stooge 1'
   'Moe':   PRINT, 'Stooge 2'
   'Curly': PRINT, 'Stooge 3'
ELSE: PRINT, 'Not a Stooge'
ENDCASE
```

Another example shows the CASE statement with the number 1 as the selector expression of the CASE. One is equivalent to *true* and is matched against each of the conditionals.

```
CASE 1 OF
   (X GT 0) AND (X LE 50): Y = 12 * X + 5
```

```
      (X GT 50) AND (X LE 100): Y = 13 * X + 4
      (X LE 200): BEGIN
        Y = 14 * X - 5
        Z = X + Y
      END
    ELSE: PRINT, 'X has an illegal value.'
    ENDCASE
```

In this CASE statement, only one clause is selected, and that clause is the first one whose value is equal to the value of the case selector expression.

**Tip** ————————————————————————————————————————————————

Each clause is tested in order, so it is most efficient to order the most frequently selected clauses first.

# SWITCH

The SWITCH statement is used to select one statement for execution from multiple choices, depending upon the value of the expression following the word SWITCH. This expression is called the switch selector expression.

The general form of the SWITCH statement is as follows:

```
SWITCH Expression OF
   Expression: Statement
   ...
   Expression: Statement
[ELSE: Statement ]
ENDSWITCH
```

Each statement that is part of a SWITCH statement is preceded by an expression that is compared to the value of the selector expression. SWITCH executes by comparing the SWITCH expression with each selector expression in the order written. If a match is found, program execution jumps to that statement and execution continues from that point. Unlike the CASE statement, execution does not resume below the SWITCH statement after the matching statement is executed. Whereas CASE executes at most one statement within the CASE block, SWITCH executes the first matching statement and any following statements in the SWITCH block. Once a match is found in the SWITCH block, execution falls through to any remaining statements. For this reason, the BREAK statement is commonly used within SWITCH statements to force an immediate exit from the SWITCH block.

The ELSE clause of the SWITCH statement is optional. If included, it matches any selector expression, causing its code to be executed. For this reason, it is usually written as the last clause in the switch statement. The ELSE statement is executed only if none of the preceding statement expressions match. If an ELSE clause is not included and none of the values match the selector, program execution continues immediately below the SWITCH without executing any of the SWITCH statements.

# CASE Versus SWITCH

The CASE and SWITCH statements are similar in function, but differ in the following ways:

•   Execution exits the CASE statement at the end of the matching statement. By contrast, execution within a SWITCH statement falls through to the next statement. The following table illustrates this difference:

| CASE | SWITCH |
|------|--------|
| ```
x=2
CASE x OF
   1: PRINT, 'one'
   2: PRINT, 'two'
   3: PRINT, 'three'
   4: PRINT, 'four'
ENDCASE
``` | ```
x=2
SWITCH x OF
   1: PRINT, 'one'
   2: PRINT, 'two'
   3: PRINT, 'three'
   4: PRINT, 'four'
ENDSWITCH
``` |
| IDL Prints: <br> ``` two ``` | IDL Prints: <br> ``` two three four ``` |

*Table 7-2:  CASE versus SWITCH*

Because of this difference, the BREAK statement is often used within SWITCH statements, but less frequently within CASE. (For more information on using the BREAK statement, see "BREAK" on page 135.) For example, we can add a BREAK statement to the SWITCH example in the above table to make the SWITCH example behave the same as the CASE example:

```
x=2
SWITCH x OF
   1: PRINT, 'one'
   2: BEGIN
         PRINT, 'two'
         BREAK
      END
   3: PRINT, 'three'
   4: PRINT, 'four'
ENDSWITCH
```

IDL Prints:

```
two
```

• If there are no matches within a CASE statement and there is no ELSE clause, IDL issues an error and execution halts. Failure to match is not an error within a SWITCH statement. Instead, execution continues immediately following the SWITCH.

The decision on whether to use CASE or SWITCH comes down deciding which of these behaviors fits your code logic better. For example, our first example of the CASE statement looked like this:

```
CASE name OF
    'Larry': PRINT, 'Stooge 1'
    'Moe':   PRINT, 'Stooge 2'
    'Curly': PRINT, 'Stooge 3'
ELSE: PRINT, 'Not a Stooge'
ENDCASE
```

We could write this example using SWITCH:

```
SWITCH name OF
    'Larry': BEGIN
                PRINT, 'Stooge 1'
                BREAK
             END
    'Moe':   BEGIN
                PRINT, 'Stooge 2'
                BREAK
             END
    'Curly': BEGIN
                PRINT, 'Stooge 3'
                BREAK
             END
ELSE: PRINT, 'Not a Stooge'
ENDSWITCH
```

Clearly, this code can be more succinctly expressed using a CASE statement.

There may be other cases when the fall-through behavior of SWITCH suits your application. The following example illustrates an application that uses SWITCH more effectively. The DAYS_OF_XMAS procedure accepts an integer argument specifying which of the 12 days of Christmas to start on. It starts on the specified day, and prints the presents for all previous days. If we enter 3, for example, we want to print the presents for days 3, 2, and 1. Therefore, the fall-through behavior of SWITCH fits this problem nicely. The first day of Christmas requires special handling, so we use a BREAK statement at the end of the statement for case 2 to prevent execution of the statement associated with case 1.

```
PRO DAYS_OF_XMAS, day

   IF (N_ELEMENTS(day) EQ 0) THEN DAY = 12
   IF ((day LT 1) OR (day GT 12)) THEN day = 12
   day_name = [ 'First', 'Second', 'Third', 'Fourth', 'Fifth', $
                'Sixth', 'Seventh', 'Eighth', 'Ninth', 'Tenth',$
                'Eleventh', 'Twelfth' ]

   PRINT, 'On The ', day_name[day - 1], $
      ' Day Of Christmas My True Love Gave To Me:'

   SWITCH day of
      12: PRINT, '    Twelve Drummers Drumming'
      11: PRINT, '    Eleven Pipers Piping'
      10: PRINT, '    Ten Lords A-Leaping'
       9: PRINT, '    Nine Ladies Dancing'
       8: PRINT, '    Eight Maids A-Milking'
       7: PRINT, '    Seven Swans A-Swimming'
       6: PRINT, '    Six Geese A-Laying'
       5: PRINT, '    Five Gold Rings'
       4: PRINT, '    Four Calling Birds'
       3: PRINT, '    Three French Hens'
       2: BEGIN
             PRINT, '    Two Turtledoves'
             PRINT, '    And a Partridge in a Pear Tree!'
             BREAK
           END
       1: PRINT, '    A Partridge in a Pear Tree!'
   ENDSWITCH
END
```

If we pass the value 3 to the DAYS_OF_XMAS procedure, we get the following
output. Achieving this behavior with CASE would be difficult.

```
On The Third Day Of Christmas My True Love Gave To Me:
    Three French Hens
    Two Turtledoves
    And a Partridge in a Pear Tree!
```

# FOR...DO

The FOR statement is used to execute one or more statements repeatedly, while incrementing or decrementing a variable with each repetition, until a condition is met. It is analogous to the DO statement in FORTRAN.

In IDL, there are two types of FOR statements: one with an implicit increment of 1 and the other with an explicit increment. If the condition is not met the first time the FOR statement is executed, the subject statement is not executed. See the following topics for details:

- "FOR Statement with an Increment of One" on page 127
- "FOR Statement with Variable Increment" on page 130
- "Sequence of the FOR Statement" on page 131

## Avoid Invariant Expressions

When using FOR loops, you can increase program efficiency by avoiding invariant expressions. Expressions whose values do not change inside a loop should be moved outside the loop. For example, in the loop:

```
FOR I = 0, N - 1 DO arr[I, 2*J-1] = ...,
```

the expression (2*J-1) is invariant and should be evaluated only once before the loop is entered:

```
temp = 2*J-1
FOR I = 0, N-1 DO arr[I, temp] = ....
```

See Chapter 15, "Arrays" for details on working with arrays.

## FOR Statement with an Increment of One

The FOR statement with an implicit increment of one is written as follows:

```
FOR Variable = Expression, Expression DO Statement
```

The variable after the FOR is called the index variable and is set to the value of the first expression. The subject statement is executed, and the index variable is incremented by 1 until the index variable is larger than the second expression. This second expression is called the limit expression. Complex limit and increment expressions are converted to floating-point type.

**Warning**

The data type of the index variable is determined by the type of the initial value expression. Keep this fact in mind to avoid the following:

```
FOR I = 0, 50000 DO ... ...
```

This loop does not produce the intended result. Converting the longword constant 50,000 to a short integer yields −15,536 because of truncation. The loop is not executed. The index variable's initial value is larger than the limit variable. The loop should be written as follows:

```
FOR I = 0L, 50000 DO ... ...
```

Note also that changing the data type of an index variable within a loop is not allowed, and will cause an error.

**Warning**

Also be aware of FOR loops that are entered but are not terminated after the expected number of iterations, because of the truncation effect. For example, if the index value exceeds the maximum value for the initial data type (and so is truncated) when it is expected instead to exceed the specified index limit, then the loop will continue beyond the expected number of iterations.

The following FOR statement continues infinitely:

```
FOR i = 0B, 240, 16 DO PRINT, i
```

The problem occurs because the variable i is initialized to a byte type with 0B. After the index reaches the limit value 240B, i is incremented by 16, causing the value to go to 256B, which is interpreted by IDL as 0B, because of the truncation effect. As a result, the FOR loop "wraps around" and the index can never be exceeded.

## Example — FOR Statement with Increment of One

A simple FOR statement:

```
FOR I = 1, 4 DO PRINT, I, I^2
```

This statement produces the following output:

```
1    1
2    4
```

```
3    9
4   16
```

The index variable I is first set to an integer variable with a value of one. The call to the PRINT procedure is executed, then the index is incremented by one. This is repeated until the value of I is greater than four at which point execution continues at the statement following the FOR statement.

The next example displays the use of a block structure (instead of a single statement) as the subject of the FOR statement. The example is a common process used for computing a count-density histogram. (Note that a HISTOGRAM function is provided by IDL.)

```
FOR K = 0, N - 1 DO BEGIN
    C = A[K]
    HIST(C) = HIST(C)+1
ENDFOR
```

The next example displays a FOR statement with floating-point index and limit expressions, where X is set to a floating-point variable and steps through the values (1.5, 2.5, ..., 10.5):

```
FOR X = 1.5, 10.5 DO S = S + SQRT(X)
```

The indexing variables and expressions can be integer, longword, floating-point, or double-precision. The type of the index variable is determined by the type of the first expression after the "=" character.

**Warning** ——————————————————————————————————————

Due to the inexact nature of IEEE floating-point numbers, using floating-point indexing can cause "infinite loops" and other problems. This problem is also manifested in both the C and FORTRAN programming languages. For example, the numbers 0.1, 0.01, 1.6, and 1.7 do not have exact representations under the IEEE standard. To see this phenomenon, enter the following IDL command:

```
PRINT, 0.1, 0.01, 1.6, 1.7, FORMAT='(f20.10)'
```

IDL prints the following *approximations* to the numbers we requested:

```
0.1000000015
0.0099999998
1.6000000238
1.7000000477
```

See "Accuracy and Floating Point Operations" on page 266 for more information about floating-point numbers.

# FOR Statement with Variable Increment

The format of the second type of FOR statement is as follows:

```
FOR Variable = Expression₁, Expression₂, Increment DO Statement
```

This form is used when an increment other than 1 is desired.

The first two expressions describe the range of numbers for the index variable. The Increment specifies the increment of the index variable. A negative increment allows the index variable to step downward.

## Example — FOR Statement with Variable Increment

The following examples demonstrate the second type of FOR statement.

```
;Decrement, K has the values 100., 99., ..., 1.
FOR K = 100.0, 1.0, -1 DO ...

;Increment by 2., loop has the values 0., 2., 4., ..., 1022.
FOR loop = 0, 1023, 2  DO ...

;Divide range from bottom to top by 4.
FOR mid = bottom, top, (top - bottom)/4.0 DO ...
```

**Warning** ─────────────────────────────────────────────────────

If the value of the increment expression is zero, an infinite loop occurs. A common mistake resulting in an infinite loop is a statement similar to the following:

```
FOR X = 0, 1, .1 DO ....
```

The variable X is first defined as an integer variable because the initial value expression is an integer zero constant. Then the limit and increment expressions are converted to the type of X, integer, yielding an increment value of zero because .1 converted to integer type is 0. The correct form of the statement is:

```
FOR X = 0., 1, .1 DO ....
```

which defines X as a floating-point variable.

────────────────────────────────────────────────────────────────────

# Sequence of the FOR Statement

The FOR statement performs the following steps:

1. The value of the first expression is evaluated and stored in the specified variable, which is called the index variable. The index variable is set to the type of this expression.

2. The value of the second expression is evaluated, converted to the type of the index variable, and saved in a temporary location. This value is called the limit value.

3. The value of the third expression, called the step value, is evaluated, type-converted if necessary, and stored. If omitted, a value of 1 is assumed.

4. If the index variable is greater than the limit value (in the case of a positive step value) the FOR statement is finished and control resumes at the next statement. Similarly, in the case of a negative step value, if the index variable is less than the limit value, control resumes after the FOR statement.

5. The statement or block following the DO is executed.

6. The step value is added to the index variable.

7. Steps 4, 5, and 6 are repeated until the test of Step 4 fails.

# REPEAT...UNTIL

REPEAT...UNIL loops are used to repetitively execute a subject statement until a condition is true. The condition is checked after the subject statement is executed. Therefore, the subject statement is always executed at least once. (See "Definition of True and False" on page 138 for details on how the "truth" of an expression is determined.)

The syntax of the REPEAT statement is as follows:

```
REPEAT statement UNTIL expression
```

or

```
REPEAT BEGIN
   statements
ENDREP UNTIL expression
```

## Examples — REPEAT...UNTIL

The following example finds the smallest power of 2 that is greater than B:

```
A = 1
B = 10
REPEAT A = A * 2 UNTIL A GT B
```

The subject statement can also be in the form of a block:

```
A = 1
B = 10
REPEAT BEGIN
   A = A * 2
ENDREP UNTIL A GT B
```

The next example sorts the elements of ARR using the inefficient bubble sort method. (A more efficient way to sort elements is to use IDL's SORT function.)

```
;Sort array.
REPEAT BEGIN
   ;Set flag to true.
   NOSWAP = 1
   FOR I = 0, N - 2 DO IF arr[I] GT arr[I + 1]THEN BEGIN
     ;Swapped elements, clear flag.
     NOSWAP = 0
     T = arr[I] & arr[I] = arr[I + 1] & arr[I + 1] = T
     ENDIF
;Keep going until nothing is moved.
ENDREP UNTIL NOSWAP
```

# WHILE...DO

WHILE...DO loops are used to execute a statement repeatedly while a condition remains true. The WHILE...DO statement is similar to the REPEAT...UNTIL statement except that the condition is checked prior to the execution of the statement. (See "Definition of True and False" on page 138 for details on how the "truth" of an expression is determined.)

The syntax of the WHILE...DO statement is as follows:

```
WHILE expression DO statement
```

or

```
WHILE expression DO BEGIN
   statements
ENDWHILE
```

When the WHILE statement is executed, the conditional expression is tested, and if it is true, the statement following the DO is executed. Control then returns to the beginning of the WHILE statement, where the condition is again tested. This process is repeated until the condition is no longer true, at which point the control of the program resumes at the next statement.

In the WHILE statement, the subject is never executed if the condition is initially false.

## Examples — WHILE...DO

The following example reads data until the end-of-file is encountered:

```
WHILE ~ EOF(1) DO READF, 1, A, B, C
```

The subject statement can also be in the form of a block:

```
WHILE ~ EOF(1) DO BEGIN
   READF, 1, A, B, C
ENDWHILE
```

The next example demonstrates one way to find the first element of an array greater than or equal to a specified value assuming the array is sorted into ascending order:

```
array = [2, 3, 5, 6, 10]
i = 0 ;Initialize index
n = N_ELEMENTS(array)

;Increment i until a point larger than 5 is found or the end of the
;array is reached:
```

```
WHILE (array[i] LT 5) AND (i LT n) DO i = i + 1

PRINT, 'The first element >= 5 is element ', i
```

IDL Prints:

```
The first element >= 5 is element    2
```

**Tip** ────────────────────────────────────────────────────────────────

Another way to accomplish the same thing is with the WHERE command, which is
used to find the subscripts of the points where ARR[I] is greater than or equal to X.

```
P = WHERE(arr GE X)
;Save first subscript:
I = P(0)
```

────────────────────────────────────────────────────────────────────────

# Jump Statements

Jump statements can be used to modify the behavior of conditional and iterative statements. Jump statements allow you to exit a loop, start the next iteration of a loop, or explicitly transfer program control to a specified location in your program.

## Statement Labels

Labels are the destinations of GOTO statements as well as the ON_ERROR and ON_IOERROR procedures. The label field is simply an identifier followed by a colon. Label identifiers, as with variable names, consist of 1 to 15 alphanumeric characters, and are case insensitive. The dollar sign ($) and underscore (_) characters can appear after the first character. Some examples of labels are as follows:

```
LABEL1:
LOOP_BACK: A = 12
I$QUIT: RETURN   ;Comments are allowed.
```

## BREAK

The BREAK statement provides a convenient way to immediately exit from a loop (FOR, WHILE, REPEAT), CASE, or SWITCH statement without resorting to the GOTO statement.

### Example

This example illustrates a situation in which using the BREAK statement makes a loop more efficient. In this example, we create a 10,000-element array of integers from 0 to 9999, ordered randomly. Then we use a loop to find where in the array the value 5 is located. If the value is found, we BREAK out of the loop because there is no need to check the rest of the array:

**Note**

This example could be written more efficiently using the WHERE function. This example is intended only to illustrate how BREAK might be used.

```
; Create a randomly-ordered array of integers
; from 0 to 9999:

array = SORT(RANDOMU(seed,10000))
n = N_ELEMENTS(array)

; Find where in array the value 5 in located:
```

```
FOR i = 0,n-1 DO BEGIN
   IF (array[i] EQ 5) THEN BREAK
ENDFOR

PRINT, i
```

We could write this loop without using the BREAK statement, but this would require us to continue the loop even after we find the value we're looking for (or resort to using a GOTO statement):

```
FOR i = 0, n-1 DO BEGIN
   IF (array[i] EQ 5) THEN found=i
ENDFOR

PRINT, found
```

# CONTINUE

The CONTINUE statement provides a convenient way to immediately start the next iteration of the enclosing FOR, WHILE, or REPEAT loop. Whereas the BREAK statement exits from a loop, the CONTINUE statement exits only from the current loop iteration, proceeding immediately to the next iteration.

**Note**

Do not confuse the CONTINUE statement described here with the .CONTINUE executive command The two constructs are not related, and serve completely different purposes.

**Note**

CONTINUE is not allowed within CASE or SWITCH statements. This is in contrast with the C language, which does allow this.

## Example

This example presents one way (not necessarily the best) to print the even numbers between 1 and 10:

```
FOR I=1,10 DO BEGIN
   IF (I AND 1) THEN CONTINUE  ; If odd, start next iteration
      PRINT, I
ENDFOR
```

# GOTO

The GOTO statement is used to transfer program control to a point in the program specified by the label. The GOTO statement is generally considered to be a poor programming practice that leads to unwieldy programs. Its use should be avoided. However, for those cases in which the use of a GOTO is appropriate, IDL does provide the GOTO statement.

Note that using a GOTO to jump into the middle of a loop results in an error.

The syntax of the GOTO statement is as follows:

```
GOTO, Label
```

**Warning** ——————————————————————————————————————————

You must be careful in programming with GOTO statements. It is not difficult to get into a loop that will never terminate, especially if there is not an escape (or test) within the statements spanned by the GOTO.

———————————————————————————————————————————————————————

## Example

In the following example, the statement at label JUMP1 is executed after the GOTO statement, skipping any intermediate statements:

```
GOTO, JUMP1
PRINT, 'Skip this' ; This statement is skipped
PRINT, 'Skip this' ; This statement is also skipped
JUMP1: PRINT, 'Do this'
```

The label can also occur before the GOTO statement that refers to the label, but you must be careful to avoid an endless loop. GOTO statements are frequently the subjects of IF statements, as in the following statement:

```
IF A NE G THEN GOTO, MISTAKE
```

# Definition of True and False

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

- `IF...THEN...ELSE` statements
- `? :` inline conditional expressions
- `WHILE...DO` statements
- `REPEAT...UNTIL` statements

The definition of *true* and *false* for the different data types is as follows:

| Data Type | True | False |
|---|---|---|
| Byte, integer, and long | Odd integers | Zero or even integers |
| Floating point and complex | Non-zero values | Zero |
| String | Any string with non-zero length | Null string (" ") |
| Heap variables (pointers and object references) | Non-null values | Null values |

*Table 7-3: Default Definitions of True and False*

If the LOGICAL_PREDICATE compile option is set:

| Data Type | True | False |
|---|---|---|
| Numerical values | Non-zero values | Zero |
| String or heap variables | Non-null values | Null values |

*Table 7-4: True and False Definitions with LOGICAL_PREDICATE*

See "COMPILE_OPT" (*IDL Reference Guide*) for additional details on the LOGICAL_PREDICATE compilation option.

In the following example, the logical statement for the condition is a conjunction of two conditions:

```
IF (LON GT -40) AND (LON LE -20) THEN ...
```

If both conditions (LON being larger than –40 and less than or equal to –20) are true, the statement following the THEN is executed.

# Chapter 8
# Debugging and Error-Handling

The following topics are covered in this chapter:

# Debugging and Error-Handling Overview

IDL provides several tooks to help find and handle errors in your code. This chapter describes debugging and error-handling features that are intrinsic to IDL itself — they are available at the IDL command prompt (either in command-line or graphical mode), or programmatically via IDL routines and statements.

**Tip** —————————————————————————————————————

The IDL Workbench provides additional debugging features that can speed debugging of your IDL code. See Debugging Tools in the IDL Workbench online documentation for a complete description.

———————————————————————————————————————————

# What Happens When Execution Stops

In the default case, whenever an error is detected by IDL during the execution of a program, program execution stops and an error message is printed. The execution context is that of the program unit (procedure, function, or main program) in which the error occurred. If you are using the IDL Workbench when execution is interrupted, the code in which the error occurred is displayed in an editor window and an indicator is placed next to the line that will be executed when processing resumes. The routine being compiled need not already be shown in an editor window. If a routine compiled with the .RUN, .RNEW, or .COMPILE executive commands contains an error, the IDL Workbench will display the file automatically.

When execution stops, you can take the following steps:

- Correct the problem and continuing program execution (see "Correcting Errors During Execution" on page 150)

- Anticipate and handle errors to avoid execution halt ("Controlling and Recovering from Errors" on page 152)

To understand what is happening during program execution, consider setting breakpoint and stepping through the code. See "Working with Breakpoints" on page 145.

## Example: Correcting Undefined Variable

A simple procedure, called BROKEN, has been included in the IDL distribution. An error occurs when BROKEN is executed. Start the IDL Workbench. Call the BROKEN procedure by entering:

```
BROKEN
```

at the IDL command line. An error is reported in the console; if you are using the IDL Workbench, an editor window displays the file BROKEN.PRO :

```
; $Id: broken.pro,v 1.1 1996/10/01 22:01:54 doug Exp $

PRO BROKEN
    PRINT, i
    PRINT, i*2
    PRINT, i*3
    PRINT, i*4
END
```

A "Variable is undefined" error has occurred. The line of code that first references the undefined variable is noted in the error message and (if applicable) highlighted with an arrow in the editor.

There are several ways to fix this error. We could edit the program file to explicitly define the variable i, or we could change the program so that it accepts a parameter at the command line. Instead, we'll define the variable i on the fly and continue execution of the program without making any changes to the program file. To define the variable i and assign it the value 10, enter at the command line:

```
i = 10
```

Next, enter

```
.CONTINUE
```

at the command line, or select **Run → Resume** in the IDL Workbench.

# Working with Breakpoints

A *breakpoint* is a marker in an IDL source code file that tells IDL to halt execution temporarily, allowing you to inspect the state of program variables in the program unit where the breakpoint occurred. Breakpoints allow you to control the flow of execution of your IDL program, stopping and starting at will.

**Note** ────────────────────────────────────────

While you can set and use breakpoints in command-line mode using the BREAKPOINT routine and various Executive Commands, breakpoints are vastly more useful when working within graphical mode.

────────────────────────────────────────

To experiment with breakpoints, do the following:

1. In the IDL Workbench, type

   ```
   .EDIT broken
   ```

   at the IDL command prompt. This loads the file broken.pro into an editor window.

2. Edit the first program line to read as follows and then save and compile the program:

   ```
   PRO BROKEN, i
   ```

   This allows you to pass a value for i to the program.

3. Set a breakpoint in broken.pro by placing the cursor in the line that reads:

   ```
   PRINT, i*2
   ```

   and selecting **Toggle Breakpoint** from the **Run** menu or simply double-clicking on the line. A blue breakpoint dot appears next to the line.

4. Now enter the following to execute the program:

   ```
   BROKEN, 10
   ```

   The Console view displays the following:

   ```
   10
   % Breakpoint at: BROKEN 10
   ```

   and a current line indicator arrow stops at the breakpoint.

   **Note** ────────────────────────────────────────

   When execution halts, you may see the **Confirm Perspective Switch** dialog.

   ────────────────────────────────────────

5.  Inspect the value of the variable i by typing

    ```
    PRINT, i
    ```

    at the command line, or by hovering the mouse pointer over the variable in the editor window.

# Stepping Through a Program

Once execution halts at a breakpoint, you can step through the program manually, or continue execution automatically. When stepping through a main program, if the next line calls another IDL procedure or function, you have three options with which to handle execution of the nested program:

- **Step Into** executes statements in order by successive .STEP commands

- **Step Over** executes statements to the end of the called function, without interactive capability

- **Step Out** to continue processing until the main program returns.

While you can step through code in command-line mode using the .STEP, .STEPOVER and .RETURN executive commands, these operations are easier and more interactive when working within graphical mode. To experiment with stepping, do the following:

1. In the IDL Workbench, type

   ```
   .EDIT broken
   ```

   at the IDL command prompt. This loads the file broken.pro into an editor window.

2. Set a breakpoint in broken.pro by placing the cursor in the line that reads:

   ```
   PRINT, i*2
   ```

   and selecting **Toggle Breakpoint** from the **Run** menu or simply double-clicking on the line. A blue breakpoint dot appears next to the line.

3. Now enter the following to execute the program:

   ```
   BROKEN, 10
   ```

   Execution stops at the specified line.

4. To step through the program, select **Step Over** from the **Run** menu, or press F6. Statements are executed one at a time. Alternately, select **Resume** from the **Run** menu or press F8 to let IDL continue until it hits another breakpoint, an error, or the end of the file.

# Monitoring Variable Values

When execution halts, there are several ways to see the values of program variables. These include:

- Check variable values from the command line — see "Showing Variable Values During Execution" below

- Use the Variable Watch window — see "The Variables View" on page 149

- Recover "missing" variables — see "Disappearing Variables" on page 149

## Showing Variable Values During Execution

When execution stops you can query the values of current variables in the program scope using the PRINT and HELP routines. For instance, suppose you have created the following program:

```
FUNCTION hello_who, who
    RETURN, 'Hello ' + who
END

PRO hello_main
    name = ''
    READ, name, PROMPT='Enter Name: '
    str = HELLO_WHO(name)
    PRINT, str
END
```

Place a breakpoint on the PRINT, str line and then compile and run the program. Enter a name at the IDL command line when prompted. When execution halts, return the value of the name variable by entering,

```
PRINT, name
```

The Console view shows the name you have entered.

Return information about the str variable by entering:

```
HELP, str
```

The Console view shows the variable name, data type and value. This information is also available in the Variables view, described in the following section.

**Tip** ─────────────────────────────────────────────

You can also place PRINT and HELP statements in your program to see variable values without pausing program execution. As these statements are encountered, values are printed to the Console.

─────────────────────────────────────────────────────

# The Variables View

The Variables view window displays the values of variables in the current execution context. If the calling context changes during execution — as when stepping into a procedure or function — the variable table is replaced with a table appropriate to the new context. See Variables view in the IDL Workbench online help for a complete description.

# Disappearing Variables

IDL users may find that all their variables have seemingly disappeared after an error occurs inside a procedure or function. The misunderstood subtlety is that after the error occurs, IDL's context is *inside the called procedure*, not in the main level. All variables in procedures and functions, with the exception of parameters and common variables, are local in scope. Typing RETURN or RETALL will make the lost variables reappear.

RETALL is best suited for use when an error is detected in a procedure and it is desired to return immediately to the main program level despite nested procedure calls. RETALL issues RETURN commands until the main program level is reached.

The HELP command can be used to see the current call stack (i.e., which program unit IDL is in and which program unit called it). For more information, see "HELP" (*IDL Reference Guide*).

# Correcting Errors During Execution

Sometimes it is possible to recover from an error by manually entering statements to correct the problem. Possibilities include setting the values of variables, closing files, etc., and then entering the command .CONTINUE, which resumes execution of the program unit at the beginning of the statement that caused the error.

As an example, if an error occurs because an undefined variable is referenced, you can simply define the variable at the command prompt and then continue execution with .CONINUE. Of course, this is a temporary solution. You should still edit the program file to fix the problem permanently.

See "Example: Correcting Undefined Variable" on page 143 for a simple example.

# Obtaining Traceback Information

It is sometimes useful for a procedure or function to obtain information about its caller(s). The SCOPE_TRACEBACK function returns a string array describing the contents of the procedure stack. The first element of the resulting array contains information for the IDL main program ($MAIN$). Each subsequent element contains information for the next routine in the call stack. The final element contains the information for the currently running routine. Each element of this array contains the module name, source filename, and line number of the routine it describes.

For example, the following code fragment prints the name of its caller, followed by the source filename and line number of the call:

```
A = SCOPE_TRACEBACK()

; Print next to last element: caller of the current routine
PRINT, 'Called from: ', A[N_ELEMENTS(A)-2]
```

This results in a message of the following form:

```
Called from: DIST </usr/local/itt/idl/lib/dist.pro (27)>
```

SCOPE_TRACEBACK can also provide more detailed information for the call stack. See "SCOPE_TRACEBACK" (*IDL Reference Guide*) for more information about the function's capabilities.

In the IDL Workbench, you can visually inspect the call stack using the Debug view.

# Controlling and Recovering from Errors

IDL divides possible execution errors into three categories: input/output, math, and all others. There are three main error-handling routines: CATCH, ON_ERROR, and ON_IOERROR. CATCH is a generalized mechanism for handling exceptions and errors. The ON_ERROR routine handles regular errors when an error handler established by the CATCH procedure is not present. The ON_IOERROR routine allows you to change the default way in which input/output errors are handled. The FINITE and CHECK_MATH routines provide control over math errors.

**Note** ─────────────────────────────────────────────────────────────

The !ERROR_STATE system variable is updated when errors occur. At the beginning of an IDL session, !ERROR_STATE contains default information. To see this information, you can either view !ERROR_STATE from the System field of the Variable Watch Window (see "The Variables View" on page 149) or you can enter PRINT, !ERROR_STATE at the Command Line. After an error has occurred, all of the fields of !ERROR_STATE display their updated status. Refer to "!ERROR_STATE" (*IDL Reference Guide*) for details.

─────────────────────────────────────────────────────────────────────

You can also write code in such a manner as to anticipate and handle potential errors, especially when you are writing your own routines. See the following topics in Chapter 5, "Creating Procedures and Functions" for details:

- "Determining Variable Scope" on page 84
- "Determining if a Keyword is Set" on page 88
- "Supplying Values for Missing Keywords" on page 89
- "Supplying Values for Missing Arguments" on page 90

## Interaction of CATCH, ON_ERROR, and ON_IOERROR

Error handlers established by calls to CATCH supersede calls to ON_ERROR. However, calls to ON_IOERROR made in the procedure that causes an I/O error supersede any error handling mechanisms created with CATCH and the program branches to the label specified by ON_IOERROR.

The following figure is a flow chart of how errors are handled in IDL.



*Figure 8-1: Error Handling in IDL*

# Creating Custom Error Messages

To generate an exception and cause control to return to the error handler, use the MESSAGE procedure. Calling MESSAGE generates an exception that sets the !ERROR_STATE system variable. !ERROR_STATE.MSG is set to the string used as an argument to MESSAGE.

The MESSAGE procedure is used by user procedures and functions to issue errors. It has the form:

```
MESSAGE, Text
```

where *Text* is a scalar string that contains the text of the error message.

The MESSAGE procedure issues error and informational messages using the same mechanism employed by built-in IDL routines. By default, the message is issued as an error, the message is output, and IDL takes the action specified by the ON_ERROR procedure.

As a side effect of issuing the error, appropriate fields of the system variable !ERROR_STATE are set; the text of the error message is placed in !ERROR_STATE.MSG, or in !ERROR_STATE.SYS_MSG for the operating system's component of the error message. See "!ERROR_STATE" (*IDL Reference Guide*) for more information.

As an example, assume the statement:

```
MESSAGE, 'Unexpected value encountered.'
```

is executed in a procedure named CALC. IDL would print:

```
% CALC: Unexpected value encountered.
```

and execution would halt.

The MESSAGE procedure accepts several keywords that modify its behavior. See "MESSAGE" (*IDL Reference Guide*) for additional details.

Another use of MESSAGE involves re-signaling trapped errors. For example, the following code uses ON_IOERROR to read from a file until an error (presumably end-of-file) occurs. It then closes the file and reissues the error.

```
; Open the data file.
OPENR, UNIT, 'DATA.DAT', /GET_LUN

; Arrange for jump to label EOD when an input/output error occurs.
ON_IOERROR, EOD

; Read every line of the file.
```

```
WHILE 1 DO READF, UNIT, LINE

; An error has occurred. Cancel the input/output error trap.
EOD: ON_IOERROR, NULL

; Close the file.
FREE_LUN, UNIT

; Reissue the error. !ERROR_STATE.MSG contains the appropriate
; text. The IOERROR keyword causes it to be issued as an
; input/output error. Use of NONAME prevents MESSAGE from tacking
; the name of the current routine to the beginning of the message
; string since !ERROR_STATE.MSG already contains it.
MESSAGE, !ERROR_STATE.MSG, /NONAME, /IOERROR
```

## Message Blocks

IDL messages include text and formatting information which, when combined with text supplied in the call to MESSAGE, provide information to the program's user about the error that occurred. For example, entering

```
MESSAGE, 'Howdy, folks'
```

at the IDL command line produces the following output:

```
% $MAIN$: Howdy, folks
% Execution halted at: $MAIN$
```

indicating that the message was issued from within the IDL $MAIN$ program.

A *message block* is a collection of messages that are loaded into IDL as a single unit. At startup, IDL contains a single internal message block named IDL_MBLK_CORE, which contains the standard messages required by the IDL system. By default, MESSAGE throws the IDL_M_USER_ERR message from the IDL_MBLK_CORE message block, producing output similar to that shown above.

Dynamically loadable modules (DLMs) usually define additional message blocks for their own needs when they are loaded. In addition, if you wish to provide something other than the default error message for your own IDL programs, you can define your own message blocks and error messages. See "DEFINE_MSGBLK" and "DEFINE_MSGBLK_FROM_FILE" (*IDL Reference Guide*) for additional details. Specify the BLOCK and NAME keywords to the MESSAGE procedure to issue a message from a message block you have defined.

# Notifying the User of Errors

The DIALOG_MESSAGE function creates a modal (blocking) dialog box that can be used to display information for the user. The dialog must be dismissed, by clicking on one of its option buttons, before execution can continue.

See "DIALOG_MESSAGE" (*IDL Reference Guide*) for details or the MEMORY routine "Examples" section in the *IDL Reference Guide* for an example of using DIALOG_MESSAGE.

# Math Errors

The detection of math errors, such as division by zero, overflow, and attempting to take the logarithm of a negative number, is hardware and operating system dependent. Some systems trap more errors than other systems. On systems that implement the IEEE floating-point standard, IDL substitutes the special floating-point values NaN and Infinity when it detects a floating point math error. (See "Special Floating-Point Values" on page 158.) Integer overflow and underflow is not detected. Integer divide by zero is detected on all platforms.

## A Note on Floating-Point Underflow Errors

Floating-point underflow errors occur when a non-zero result is so close to zero that it cannot be expressed as a normalized floating-point number. In the vast majority of cases, floating-point underflow errors are harmless and can be ignored. For more information on floating-point numbers, see "Accuracy and Floating Point Operations" on page 266

# Accumulated Math Error Status

IDL handles math errors by keeping an accumulated math error status. This status, which is implemented as a longword, contains a bit for each type of math error that is detected by the hardware. When IDL automatically checks and clears this indicator depends on the value of the system variable !EXCEPT. The CHECK_MATH function also allows you to check and clear the accumulated math error status when desired.

!EXCEPT has three possible values:

### !EXCEPT=0

Do not report exceptions.

### !EXCEPT=1

The default. Report exceptions when the IDL interpreter returns to an interactive prompt. Any math errors that occurred since the last interactive prompt (or call to CHECK_MATH) are printed in the IDL command log. A typical message looks like:

```
% Program caused arithmetic error: Floating divide by 0
```

**!EXCEPT=2**

Report exceptions after each IDL statement is executed. This setting also allows IDL to report on the program context in which the error occurred, along with the line number in the procedure. A typical message looks like:

```
% Program caused arithmetic error: Floating divide by 0
% Detected at  JUNK             3 junk.pro
```

# Special Floating-Point Values

Machines which implement the IEEE standard for binary floating-point arithmetic have two special values for undefined results: NaN (Not A Number) and Infinity. Infinity results when a result is larger than the largest representation. NaN is the result of an undefined computation such as zero divided by zero, taking the square-root of a negative number, or the logarithm of a non-positive number. In many cases, when IDL encounters the value NaN in a data set, it treats it as "missing data." The special values NaN and Infinity are also accessible in the read-only system variable !VALUES. These special operands propagate throughout the evaluation process—the result of any term involving these operands is one of these two special values.

**Note** ————————————————————————————————————

For the minimum (<) and maximum (>) operators with NaN operands, the result is undefined and may not necessarily be the special value NaN. "Mathematical Operators" on page 215 for details.

————————————————————————————————————————————

For example:

```
; Multiply NaN by 3
PRINT, 3 * !VALUES.F_NAN
```

IDL prints:

```
NaN
```

It is important to remember that the value NaN is literally not a number, and as such cannot be compared with a number. For example, suppose you have an array that contains the value NaN:

```
A = [1.0, 2.0, !VALUES.F_NAN, 3.0]
PRINT, A
```

IDL prints:

```
1.00000     2.00000     NaN     3.0000
```

If you try to select elements of this array by comparing them with a number (using the WHERE function, for example), IDL might generate an error (depending on the hardware and operating system):

```
; Print the indices of A that are not equal to 1
PRINT, WHERE( A NE 1.0 )
```

IDL prints:

```
1    2    3
% Program caused arithmetic error: Floating illegal operand
```

(Depending on your hardware and operating system, you may not see the floating-point error.)

To avoid this problem, use the FINITE function to make sure arguments to be compared are in fact valid floating-point numbers:

```
PRINT, WHERE( FINITE(A) )
```

IDL prints the indices of the finite elements of A:

```
0    1    3
```

To then print the indices of the elements of A that are both finite and not equal to 1.0, you could use the command:

```
good = WHERE( FINITE(A) )
PRINT, good[WHERE(A[good] NE 1.0)]
```

IDL prints:

```
1    3
```

Similarly, if you wanted to find out which elements of an array were *not* valid floating-point numbers, you could use a command like:

```
; Print the indices of the elements of A that are not valid
; floating-point numbers.
PRINT, WHERE( ~FINITE(A) )
```

IDL prints:

```
2
```

Note that the special value Infinity *can* be compared to a floating point number. Thus, if:

```
B = [1.0, 2.0, !VALUES.F_INFINITY]
PRINT, B
```

IDL prints:

```
1.00000      2.00000          Inf
```

and

```
PRINT, WHERE(B GT 1.0)
```

IDL prints:

```
1           2
```

You can also compare numbers directly with the special value Infinity:

```
PRINT, WHERE(B EQ !VALUES.F_INFINITY)
```

IDL prints:

```
2
```

**Note**

On Windows, using relational operators such as EQ and NE with the values infinity or NaN (Not a Number) causes an "illegal operand" error. The FINITE function's INFINITY and NAN keywords can be used to perform comparisons involving infinity and NaN values. For more information, see "FINITE" on page 851.

## The FINITE Function

Use the FINITE function to explicitly check the validity of floating-point or double-precision operands on machines which use the IEEE floating-point standard. For example, to check the result of the EXP function for validity, use the following statement:

```
;Perform exponentiation.
A = EXP(EXPRESSION)

;Print error message.
IF ~ FINITE(A) THEN PRINT, 'Overflow occurred'
```

If A is an array, use the statement:

```
IF TOTAL(FINITE(A)) NE N_ELEMENTS(A) THEN
```

## Integer Conversions

It must be stressed that when converting from floating to any of the integer types (byte, signed or unsigned short integer, signed or unsigned longword integer, or signed or unsigned 64-bit integer) if overflow is important, you must explicitly check to be sure the operands are in range. Conversions to the above types from floating point, double precision, complex, and string types do not check for overflow—they simply convert the operand to the target integer type, discarding any significant bits of information that do not fit.

When run on a Sun workstation, the program:

```
A = 2.0 ^ 31 + 2
PRINT, LONG(A), LONG(-A), FIX(A), FIX(-A), BYTE(A), BYTE(-A)
```

(which creates a floating-point number 2 larger than the largest positive longword integer), prints the following:

```
2147483647 -2147483648 -1 0 255 0
% Program caused arithmetic error: Floating illegal operand
```

This result is incorrect.

**Warning** ─────────────────────────────────────────────

No error message will appear if you attempt to convert a floating number whose absolute value is between $2^{15}$ and $2^{31}$ - 1 to short integer even though the result is incorrect. Similarly, converting a number in the range of 256 to $2^{31}$ - 1 from floating, complex, or double to byte type produces an incorrect result, but no error message. Furthermore, integer overflow is usually not detected. Your programs must guard explicitly against it.

─────────────────────────────────────────────────────────

# Chapter 9
# Building Cross-Platform Applications

The following topics are covered in this chapter:

# Overview of Cross-Platform Issues

IDL is designed as a platform-independent environment for data analysis and programming. Because of this, the vast majority of IDL's routines operate the same way no matter what type of computer system you are using. IDL's cross-platform development environment makes it easy to develop an application on one type of system for use on any system IDL supports.

Despite IDL's cross-platform nature, there *are* differences between the computers that make up a multi-platform environment. Operating systems supply resources in different ways. While IDL attempts to abstract these differences and provide a common environment for all Windows and UNIX machines, there are some cases where the discrepancies cannot be overcome. This chapter discusses aspects of IDL that you may wish to consider when developing an application that will run on multiple types of computer.

**Note** ─────────────────────────────────────────────────────────────────
This chapter is *not* an exhaustive list of differences between versions of IDL for different platforms. Rather, it covers issues you may encounter when writing cross-platform applications in IDL.
──────────────────────────────────────────────────────────────────────────

# Which Operating System is Running?

In some cases, in order to effectively take platform differences into account, your application will need to execute different code segments on different systems. Operating system and IDL version information is contained in the IDL system variable !VERSION. For example, you could use an IDL CASE statement that looks something like the following to execute code that pertains to a particular operating system family:

```
CASE !VERSION.OS_FAMILY OF
    'unix'     : Code for Unix
    'Windows'  : Code for Windows
ENDCASE
```

Writing conditional IDL code based on platform information should be a last resort, used only if you cannot accomplish the same task in a platform-independent manner.

## Operating System Access

While IDL provides ways to interact with each operating system under which it runs, it is not generally useful to use operating-system native functions in a cross-platform IDL program. If you find that you must use operating-system native features, be sure to determine the current operating system (as described above) and branch your code accordingly.

# File and Path Specifications

Different operating systems use different path specification syntax and directory separation characters. The following table summarizes the different characters used by different operating systems; see "!PATH" (*IDL Reference Guide*) for further details on path specification.

| Operating System | Directory Separator | Path Element Separator |
|---|---|---|
| UNIX | / (forward slash) | : (colon) |
| Windows | \ (backward slash) | ; (semicolon) |

*Table 9-1: Directory and Path Element Separator Characters*

As a result of these differences, specifying filenames and paths explicitly in your IDL application can cause problems when moving your application to a different platform. You can effectively isolate your IDL programs from platform-specific file and path specification issues by using the FILEPATH, PATH_SEP, and DIALOG_PICKFILE functions.

## Choosing Files at Runtime

To allow users of your application to choose a file at runtime, use the DIALOG_PICKFILE function. DIALOG_PICKFILE will always return the file path with the correct syntax for the current platform. Other methods (such as reading a file name from a text field in a widget program) may or may not provide a proper file path.

## Selecting Files Programmatically

To give your application access to a file you know to be installed on the host, use the FILEPATH function. By default, FILEPATH allows you to select files that are included in the IDL distribution tree. Chances are, however, that a file you supply as part of your own application is *not* included in the IDL tree. You can still use FILEPATH by explicitly specifying the root of the directory tree to be searched.

*Figure 9-1: A Possible Directory Hierarchy for an IDL Application*

For example, suppose your application is installed in a subdirectory named MYAPP
of the root directory of the filesystem that contains the IDL distribution. You could
use the FILEPATH function and set the ROOT_DIR keyword to the root directory of
the filesystem, and use the SUBDIRECTORY keyword to select the MYAPP
directory. If you are looking for a file named myapp.dat, the FILEPATH command
looks like this:

```
file = FILEPATH('myapp.dat', ROOT_DIR=root, SUBDIR='MYAPP')
```

The problem that remains is how to specify the value of *root* properly on each
platform. This is one case where it is very difficult to avoid writing some platform-
specific code. We could write an IDL CASE statement each time the FILEPATH
function is used. Instead, the following code segment sets an IDL variable to the
string value of the root of the filesystem, and passes that variable to the ROOT_DIR
keyword. The CASE statement looks like this:

```
CASE !VERSION.OS_FAMILY OF
   'unix'    : rootdir = '/'
   'Windows' : rootdir = STRMID(!DIR, 0, 2)
ENDCASE
file = FILEPATH('myapp.dat', ROOT=rootdir, SUBDIR='MYAPP')
```

Note that the root directory under Unix is well defined, whereas the root directory on
a machine running Microsoft Windows must be determined by parsing the IDL
system variable !DIR. Under Windows, the root is assumed to be the drive letter of
the hard drive and the following colon — usually "C:".

# Files and I/O

IDL's file input and file output routines are designed to work identically on all platforms, where possible. In the case of basic operations, such as opening a text file and reading its contents, importing an image format file into an IDL array, or writing ASCII data to a file on a hard disk, IDL's I/O routines work the same way on all platforms. In more complicated cases, however, such as reading data stored in binary data format files, different operating systems may use files that are structured differently, and extra care may be necessary to ensure that IDL reads or writes files in the proper way.

Before attempting to write a cross-platform IDL application that uses more than basic file I/O, you should read and understand the sections in Chapter 18, "Files and Input/Output" that apply to the platforms your application will support. The following are a few topics to think about when writing IDL applications that do input/output.

## Byte Order Issues

Computer systems on which IDL runs support two ways of ordering the bytes that make up an arbitrary scalar: *big endian*, in which multiple byte numbers are stored in memory beginning with the most significant byte, and *little endian*, in which numbers are stored beginning with the least significant byte. The following table lists the processor types and operating systems IDL supports and their byte ordering schemes:

| Processor Type | Operating System | Byte Ordering |
|---|---|---|
| AMD | Linux | little-endian |
|  | Windows | little-endian |
| Intel x86 | Linux | little-endian |
|  | Windows | little-endian |
|  | Macintosh OS X | little-endian |
| Motorola PowerPC | Macintosh OS X | big-endian |
| Sun SPARC | Solaris | big-endian |

*Table 9-2: Byte Ordering Schemes Used by Platforms that Support IDL*

The IDL routines BYTEORDER and SWAP_ENDIAN allow you to convert numbers from big endian format to little endian format and *vice versa*. It is often easier, however, to use the XDR (for eXternal Data Representation) format to store data that you know will be used by multiple platforms. XDR files write binary data in a standard "canonical" representation; as a result, the files are slightly larger than pure binary data files. XDR files can be read and written on any platform that supports IDL. XDR is discussed in detail in "Portable Unformatted Input/Output" on page 456.

# Math Exceptions

The detection of math errors, such as division by zero, overflow, and attempting to take the logarithm of a negative number, is hardware and operating system dependent. Some systems trap more errors than other systems. Beginning with version 5.1, IDL uses the IEEE floating-point standard on all supported systems. As a result, IDL always substitutes the special floating-point values NaN and Infinity when it detects a math error. (See "Special Floating-Point Values" on page 158 for details on NaN and Infinity.)

For information on debugging math errors, see "Math Errors" on page 157.

# Responding to Screen Size and Colors

The usability of your application may depend on responding to settings on the user's system.

## Finding Screen Size

Use the GET_SCREEN_SIZE function to determine the size of the screen on which your application is displayed. Writing code that checks the screen size allows your application to handle different screen sizes gracefully.

## Number of Colors Available

Use the N_COLORS and TABLE_SIZE fields of the !D system variable to determine the number of colors supported by the display and the number of color-table entries available, respectively.

Make sure that your application handles relatively small numbers of colors (less than 256, say) gracefully. For example, Microsoft Windows reserves the first 20 colors out of all the available colors for its own use. These colors are the ones used for title bars, window frames, window backgrounds, scroll bars, etc. If your application is running on a Windows machine with a 256-color display, it will have at most 236 colors available to work with.

Similarly, make sure that your application handles TrueColor (24-bit or 32-bit color) displays as well. If your application uses IDL's color tables, for example, you will need to force the application into 8-bit mode using the command

```
DEVICE, DECOMPOSED=0
```

to use indexed-color mode on a machine with a TrueColor display.

# Printing

IDL displays operating-system native dialogs using the DIALOG_PRINTJOB and DIALOG_PRINTERSETUP functions. Since the dialogs that control printing and printer setup differ between systems, so do the options and capabilities presented via IDL's print dialogs. If your IDL application uses IDL's printing dialogs, make sure that your interface calls the dialog your user will expect for the platform in question.

# SAVE and RESTORE

If you distribute your application via IDL SAVE files, remember that files containing IDL routines are not necessarily compatible between IDL releases. Always save your original code and re-save when a new version of IDL is released. SAVE files containing data are always compatible between releases of IDL.

**Note**

If you are restoring a file created with VAX IDL version 1, you must restore on a machine running VMS.

# Widgets in Cross-Platform Programs

IDL's user interface toolkit is designed to provide a "native" look and feel to widget-based IDL applications. Where possible, widget toolkit elements are built around the operating system's native dialogs and controls; as a result, there are instances where the toolkit behaves differently from operating system to operating system. This section describes a number of platform-dependencies in the IDL widget toolkit. Consult the descriptions of the individual DIALOG and WIDGET routines in the *IDL Reference Guide* for complete details.

## Dialog Routines

IDL's DIALOG_ routines (DIALOG_PICKFILE, etc.) rely on operating system native dialogs for most of their functionality. This means, for example, that when you use DIALOG_PICKFILE in an IDL application, Windows users will see the Windows-native file selection dialog and Motif users will see the Motif file selection dialog. Consult the descriptions of the individual DIALOG routines in the *IDL Reference Guide* for notes on the platform dependencies.

## Base Widgets

Base widgets (created with the WIDGET_BASE routine) play an especially important role in creating widget-based IDL applications because their behavior controls the way the application and its components are iconized, layered, and destroyed. See "Iconizing, Layering, and Destroying Groups of Top-Level Bases" under "WIDGET_BASE" (*IDL Reference Guide*) for details about the platform-dependent behavior.

## Positioning Widgets within a Base Widget

The widget geometry management keywords to the WIDGET_BASE routine allow a great deal of flexibility in positioning child widgets within a base widget. When building cross-platform applications, however, making use of IDL's explicit positioning features can be counterproductive.

Because IDL attempts to provide a platform-native look on each platform, widgets depend on the platform's current settings for font, font size, and "window dressing" (things like the thickness of borders and three-dimensional appearance of controls). As a result of the platform-specific appearance of each widget, attempting to position individual widgets manually within a base will seldom give satisfactory results on all platforms.

Instead, insert widgets inside base widgets that have the ROW or COLUMN keywords set, and let IDL determine the correct geometry for the current platform automatically. You can gain a finer degree of control over the layout by placing groups of widgets within sub-base widgets (that is, base widgets that are the children of other base widgets). This allows you to control the column or row layout of small groups of widgets within the larger base widget.

In particular, refrain from using the X/YSIZE and X/YOFFSET keywords in cross-platform applications. Using the COLUMN and ROW keywords instead will cause IDL to calculate the proper (platform-specific) size for the base widget based on the size and layout of the child widgets.

# Fonts Used in Widget Applications

You can specify the font used in a widget via the FONT keyword. In general, the default fonts used by IDL widgets will most closely approximate the look of a platform-native application. If you choose to specify the fonts used in your widget application, however, note that the different platforms have different font-naming schemes for device fonts. While device fonts will provide the best performance for your application, specifying device fonts for your widgets requires that you write platform-dependent code as described in "Which Operating System is Running?" on page 165. You can avoid the need for platform-dependent code by using the TrueType fonts supplied with IDL; there may be a performance penalty when the fonts are initially rendered. See Appendix H, "Fonts" (*IDL Reference Guide*) for details.

# Motif Resources

Use the RESOURCE_NAME keyword to apply standard X Window System resources to a widget on a Motif system. Resources specified via the RESOURCE_NAME keyword will be quietly ignored on Windows systems. See "RESOURCE_NAME" under "WIDGET_BASE" (*IDL Reference Guide*) for details. In general, you should not expect to be able to duplicate the level of control available via X Window System resources on other platforms.

# WIDGET_STUB

On Motif platforms, you can use the WIDGET_STUB routine to include widgets created outside IDL (that is, with the Motif widget toolkit) in your IDL applications. The WIDGET_STUB mechanism is only available under Unix, and is thus not suitable for use in cross-platform applications that will run under Microsoft Windows. WIDGET_STUB is described in the *External Development Guide*.

# Widget Event Inconsistencies

Different windowing systems provide different types of events when graphical items are displayed and manipulated. IDL attempts to provide consistent functionality on all windowing systems, but is not always completely successful. For example, enter/exit tracking events are not generated by some windowing systems. IDL attempts to provide appropriate enter/exit events, but behaviors may differ on different platforms.

Handle individual widget events carefully, and be sure to test your code on all platforms supported by your application.

# Using External Code

The use of programs written in languages other than IDL—either by calling code from an IDL program via CALL_EXTERNAL or LINKIMAGE or via the callable IDL mechanism—is an inherently platform-dependent process. Writing a cross-platform IDL program that uses CALL_EXTERNAL or LINKIMAGE requires that you provide the appropriate programs or shared libraries for each platform your application will support, and is beyond the scope of this chapter. Similarly, the Callable IDL mechanism is necessarily different from platform to platform. See the *External Development Guide* for details on writing and using external code along with IDL.

# IDL DataMiner Issues

The IDL DataMiner provides a platform-independent interface to IDL's Open
Database Connectivity (ODBC) features. Note, however, that the ODBC drivers that
allow connection to different databases are platform-dependent, and may require
platform-dependent coding. In addition, the dialogs called by the
DIALOG_DBCONNECT function are provided by the specific ODBC driver in use,
and will be different from data source to data source.

# Chapter 10
# Multithreading in IDL

This chapter describes the implementation of the IDL Thread Pool and how it can be used to accelerate your computations.

# The IDL Thread Pool

On computer systems that have more than one central processing unit, *multi-threading* can be used to increase the speed of numeric calculations by using multiple system processors to simultaneously carry out different parts of the computation. In a multithreaded environment, each *thread* handles a portion of the overall task; if several threads can run in parallel, the computation can often be completed more quickly than if the different portions of the task ran in series.

IDL's *thread pool* — a pool of computation threads that are used as helpers to accelerate numerical computations — allows for multithreading when multiple CPUs are present. IDL automatically evaluates all computations performed by routines that may benefit from multithreading to determine whether or not to use the thread pool in the current computation. This decision is based on attributes such as the number of data elements involved, the availability of multiple CPUs, and the availability of a multithreaded implementation of the algorithm in use. You can alter the parameters used by IDL to make this decision, either on a global basis for the duration of a single IDL session, or for an individual computation.

**Note** ────────────────────────────────────────────────────────────────

Multithreading does not offer the possibility of increased execution speed for all IDL routines. For a list of the routines that have been implemented to use multithreading when possible, see "Routines that Use the Thread Pool" on page 189.

────────────────────────────────────────────────────────────────────────

## Benefits of the IDL Thread Pool

The IDL thread pool will increase processing performance on certain computations. When not involved in a calculation, the threads in the thread pool are inactive and consume little in the way of system resources. When IDL encounters a computation that can use the thread pool and which would benefit from parallel execution, it divides the task into sub-parts for each thread, enables the thread pool to do the computation, waits until the thread pool completes, and then continues. Other than the improved performance, the end result is virtually indistinguishable when compared to the same computation performed in the standard single-threaded manner.

# Possible Drawbacks to the Use of the IDL Thread Pool

There are instances when allowing IDL to use its default thread pool settings can lead to undesired results. In some instances, a multithreaded implementation using the thread pool may actually take longer to complete a given job than a single-threaded implementation. If a computation uses the thread pool in an inappropriate situation, there may be other undesirable effects. The following are some situations in which the default thread pool settings may provide less than optimal results.

## Computation of a Relatively Small Number of Data Elements

Use of the IDL thread pool requires a small fixed overhead when compared to a non-threaded version of the same computation. Normally, computational speed increases when multiple CPUs work in parallel, and the speed-up is much larger than the loss due to thread pool overhead. However, if the computation does not include a large enough number of data elements (each element being a data value of a particular data type), the loss due to thread pool overhead can exceed the benefit and the overall computation speed can be slower.

To prevent the use of the thread pool for computations that involve too few data elements, IDL supports a minimum threshold value for thread pool computations. The minimum threshold value is contained in the TPOOL_MIN_ELTS field of the !CPU system variable. See the following sections for details on modifying this value.

## Large Computation that Requires Virtual Memory Use

If a computation is too large to fit into physical memory, the threads in the thread pool may cause *page faults* that will activate the virtual memory system. If more than one thread encounters this situation simultaneously, the threads will compete with each other for access to memory and performance will fall below that of a single-threaded approach to the computation.

To prevent the use of the thread pool for computations that involve too many data elements, IDL supports a maximum threshold value for thread pool computations. The maximum threshold value is contained in the TPOOL_MAX_ELTS field of the !CPU system variable. See the following sections for details on modifying this value.

## Multiple Users Competing for CPU Resources

On a large multi-user system, an IDL application that uses the thread pool may consume all available CPUs, thus affecting other users of the system by reducing overall performance.

To prevent the use of all system processors by routines that use the thread pool, IDL allows you to specify explicitly the number of CPUs that should be used in calculations that involve the thread pool. The number of processors to be used for thread pool operations is contained in the TPOOL_NTHREADS field of the !CPU system variable. See the following sections for details on modifying this value.

**Note** ─────────────────────────────────────────────────────────────────────
To change the default number of threads used by IDL, set the IDL_CPU_TPOOL_NTHREADS preference. For more information, see "!CPU Settings Preferences" (Appendix E, *IDL Reference Guide*).
─────────────────────────────────────────────────────────────────────────────

## Sensitivity to Numerical Precision

Algorithms that are sensitive to the order of operations may produce different results when performed by the thread pool. Such results are due to the use of finite precision floating point types, and are equally correct within the precision of the data type.

# Controlling the IDL Thread Pool

IDL allows you to programmatically control the use of thread pool. This section discusses the following aspects of thread pool use:

- Viewing the Current Thread Pool Settings
- Using the Default Thread Pool Settings
- Changing Global Thread Pool Settings
- Changing Thread Pool Settings for a Specific Computation
- Disabling the Thread Pool

**Note**

Multithreading does not offer the possibility of increased execution speed for all IDL routines. For a list of the routines that have been implemented to use multithreading when possible, see "Routines that Use the Thread Pool" on page 189.

## Viewing the Current Thread Pool Settings

The current values of the parameters that control IDL's use of the thread pool for computations are always available in the read-only !CPU system variable. !CPU is initialized by IDL at startup with default values for the number of CPUs (threads) to use, as well as the minimum and maximum number of data elements. To view the settings, use the following command:

```
HELP, /STRUCTURE, !CPU
```

The values of the fields in the !CPU system variable are explained in "!CPU" (*IDL Reference Guide*).

## Using the Default Thread Pool Settings

If you have more than one processor on your system, if the routine you are using is able to use the thread pool, and if the number of data elements in your computation falls into the allowed range (neither too few nor too many), then IDL will employ the thread pool in that calculation.

If the above requirements are met, IDL will automatically use the thread pool for the computation. You do not need to do anything special to enable IDL's multithreading capabilities.

# Changing Global Thread Pool Settings

Unless they are overridden by thread pool keywords supplied at the time of execution, the values contained in the !CPU system variable control IDL's use of the thread pool. !CPU is a "read-only" system variable, which means that you cannot assign values to its structure fields directly, either at the command line or within a program. However, you can set the default number of threads prior to starting IDL by using the IDL_CPU_TPOOL_NTHREADS preference. See "!CPU Settings Preferences" (Appendix E, *IDL Reference Guide*) for details. You can also change the values of the !CPU system variable for the duration of the current IDL session by using the CPU procedure.

The CPU procedure accepts the following keywords:

### TPOOL_MAX_ELTS

Set this keyword to a non-zero value to set the maximum number of data elements involved in a computation that uses the thread pool. If the number of elements in the computation exceeds the number you specify, IDL will not use the thread pool for the computation. Setting this value to 0 removes any limit on maximum number of elements, and any computation with at least TPOOL_MIN_ELTS will use the thread pool.

This keyword changes the value returned by !CPU.TPOOL_MAX_ELTS.

### TPOOL_MIN_ELTS

Set this keyword to a non-zero value to set the minimum number of data elements involved in a computation that uses the thread pool. If the number of elements in the computation is less than the number you specify, IDL will not use the thread pool for the computation. Use this keyword to prevent IDL from using the thread pool on tasks that are too small to benefit from it.

This keyword changes the value returned by !CPU.TPOOL_MIN_ELTS.

### TPOOL_NTHREADS

Set this keyword to the number of threads IDL should use when performing computations that take advantage of the thread pool. By default, IDL will use !CPU.HW_NCPU threads, so that each thread will have the potential to run in parallel with the others. Set this keyword equal to 0 (zero) to ensure that !CPU.HW_NCPU threads will be used. Set this keyword equal to 1 (one) to disable use of the thread pool.

This keyword changes the value returned by !CPU.TPOOL.NTHREADS.

**Note** ───────────────────────────────────────────────

For numerical computation, there is no benefit to using more threads than your system has CPUs. However, depending on the size of the problem and the number of other programs running on the system, there may be a performance advantage to using *fewer* CPUs. See "Possible Drawbacks to the Use of the IDL Thread Pool" on page 181 for a discussion of the circumstances under which using fewer than the maximum number of CPUs makes sense.

───────────────────────────────────────────────────────────

For more information on the CPU procedure, see "CPU" (*IDL Reference Guide*).

## Examples

The following examples illustrate use of the CPU procedure to modify IDL's global thread pool settings.

**Note** ───────────────────────────────────────────────

The following examples are designed for systems with more than one processor. The examples will generate correct results on single-processor systems, but may run more slowly than the same operations performed without the thread pool.

───────────────────────────────────────────────────────────

### Example 1

As a first example, imagine that we want to ensure that the thread pool is not used unless there are at least 50,000 data elements. We set the minimum to 50,000 since we know, for our system, that at least 50,000 floating point data elements are required before the use of the thread pool will exceed the overhead required to use it.

In addition, we want to ensure that the thread pool is not used if a calculation involves more than 1,000,000 data elements. We set the maximum to 1,000,000 since we know that 1,000,000 floating point data elements will exceed the maximum amount of memory available for the computation, requiring the use of virtual memory.

The following IDL statements use the CPU procedure to modify the minimum and maximum number of elements used in thread pool computations, create an array of floating-point values, and perform a computation on the array:

```
; Modify the thread pool settings
CPU, TPOOL_MAX_ELTS = 1000000, TPOOL_MIN_ELTS = 50000

; Create 65,341 elements of floating point data
theta = FINDGEN(361, 181)

; Perform computation
sineSquared = 1. - (COS(!DTOR*theta))^2
```

In this example, the thread pool will be used since we are performing a computation on an array of 361 x 181 = 65,341 data elements, which falls between the minimum and maximum thresholds. Note that we altered the *global* thread pool parameters in such a way that the computation was allowed. The values set by the CPU procedure will remain in effect, either until they are changed again by another call to CPU or until the end of the IDL session. An alternative approach that does not change the global defaults in shown in "Changing Thread Pool Settings for a Specific Computation" on page 187.

### Example 2

In this example, we will:

1. Save the current thread pool settings from the !CPU system variable.

2. Modify the thread pool settings so that IDL is configured, for our particular system, to efficiently perform a floating point computation.

3. Perform several floating point computations.

4. Modify the thread pool settings so that IDL is configured, for our particular system, to efficiently perform a double precision computation.

5. Perform several double precision computations.

6. Restore the thread pool settings to their original values.

The first computation will use the thread pool since it does not exceed any of the specified parameters. The second computation, since it exceeds the maximum number of data elements, will not use the thread pool.

```
; Retrieve the current thread pool settings
threadpool = !CPU

; Modify the thread pool settings
CPU, TPOOL_MAX_ELTS = 1000000, TPOOL_MIN_ELTS = 50000, $
   TPOOL_NTHREADS = 2

; Create 65,341 elements of floating point data
theta = FINDGEN(361, 181)

; Perform computations, using 2 threads
sineSquared = 1. - (COS(!DTOR*theta))^2
next computation
next computation
etc.

; Modify thread pool settings for new data type
CPU, TPOOL_MAX_ELTS = 50000, TPOOL_MIN_ELTS = 10000
```

```
; Create 65,341 elements of double precision data
theta = DINDGEN(361, 181)

; Perform computation
sineSquared = 1. - (COS(!DTOR*theta))^2
next computation
next computation
etc.

;Return thread pool settings to their initial values
CPU, TPOOL_MAX_ELTS = threadpool.TPOOL_MAX_ELTS, $
   TPOOL_MIN_ELTS = threadpool.TPOOL_MIN_ELTS, $
   TPOOL_NTHREADS = threadpool.HW_NCPU
```

Again, in this example we altered the *global* thread pool parameters. In cases where you plan to perform multiple computations that take advantage of the same thread pool configuration, changing the global thread pool parameters is convenient. In cases where only a single computation uses the specified thread pool configuration, it is easier to use the thread pool keywords to the routine that performs the computation, as described in the following section.

# Changing Thread Pool Settings for a Specific Computation

All routines that have been implemented to use the thread pool accept keywords that allow you to override the thread pool settings stored in !CPU for a single invocation of the routine. This allows you to modify the settings for a particular computation without affecting the global default settings of your session. For a list of the routines that have been implemented to use multithreading when possible, see "Routines that Use the Thread Pool" on page 189. In the *IDL Reference Guide*, documentation for routines that use the thread pool includes a section titled "Thread Pool Keywords."

The thread pool keywords are:

### TPOOL_MAX_ELTS

Set this keyword to a non-zero value to set the maximum number of data elements involved in a computation that uses the thread pool. If the number of elements in the computation exceeds the number you specify, IDL will not use the thread pool for the computation. Setting this value to 0 removes any limit on the maximum number of elements, and any computation with at least TPOOL_MIN_ELTS will use the thread pool.

This keyword overrides the default value, given by !CPU.TPOOL_MAX_ELTS.

### TPOOL_MIN_ELTS

Set this keyword to a non-zero value to set the minimum number of data elements involved in a computation that uses the thread pool. If the number of elements in the computation is less than the number you specify, IDL will not use the thread pool for the computation. Use this keyword to prevent IDL from using the thread pool on tasks that are too small to benefit from it.

This keyword overrides the default value, given by !CPU.TPOOL_MIN_ELTS.

### TPOOL_NOTHREAD

Set this keyword to explicitly prevent IDL from using the thread pool for the current computation. If this keyword is set, IDL will use the non-threaded implementation of the routine even if the current settings of the !CPU system variable would allow use of the threaded implementation.

## Example

We can use the TPOOL_MIN_ELTS and TPOOL_MAX_ELTS keywords to the COS function to modify the example used in the previous section so that our changes to the thread pool settings do not alter the global default.

```
; Create 65,341 elements of floating point data
theta = FINDGEN(361, 181)

; Perform computation and override session settings for maximum
; and minimum number of elements
sineSquared = 1. - (COS(!DTOR*theta, TPOOL_MAX_ELTS = 1000000, $
   TPOOL_MIN_ELTS = 50000))^2
```

# Disabling the Thread Pool

There are two ways to disable the thread pool in IDL:

- Use the CPU procedure to alter the global thread pool parameters.

- Use the TPOOL_NOTHREAD keyword to a routine to disable the thread pool for a specific single computation.

In the first example, we will disable the thread pool for the session by setting the number of threads to use to one:

```
CPU, TPOOL_NTHREADS = 1
```

In the next example, we will disable the thread pool for a specific computation using the TPOOL_NOTHREAD keyword:

```
sineSquared = 1. - (COS(!DTOR*theta, /TPOOL_NOTHREAD))^2
```

# Routines that Use the Thread Pool

Multithreading does not offer the possibility of increased execution speed for all IDL routines. The operators and routines currently using the thread pool in IDL are listed below, grouped by functional category.

## Binary and Unary Operators:

| | | |
|---|---|---|
| - | -- | + |
| ++ | NOT | AND |
| / | * | EQ |
| NE | GE | LE |
| GT | LT | > |
| < | OR | XOR |
| ^ | MOD | # |
| ## | | |

**Note** —————————————————————————
If an operator uses the thread pool, any compound assignment operator based on that operator (+=, *=, *etc.*) also uses the thread pool.

## Mathematical Routines:

| | | |
|---|---|---|
| • ABS | • ERRORF | • MATRIX_MULTIPLY |
| • ACOS | • EXP | • PRODUCT |
| • ALOG | • EXPINT | • ROUND |
| • ALOG10 | • FINITE | • SIN |
| • ASIN | • FLOOR | • SINH |
| • ATAN | • GAMMA | • SQRT |
| • CEIL | • GAUSSINT | • TAN |

- CONJ
- COS
- COSH
- IMAGINARY
- ISHFT
- LNGAMMA
- TANH
- VOIGT

## Image Processing Routines:

- BYTSCL
- CONVOL
- FFT
- INTERPOLATE
- POLY_2D
- TVSCL

## Array Creation Routines:

- BINDGEN
- BYTARR
- CINDGEN
- DCINDGEN
- DCOMPLEXARR
- DINDGEN
- FINDGEN
- INDGEN
- LINDGEN
- L64INDGEN
- MAKE_ARRAY
- REPLICATE
- UINDGEN
- ULINDGEN
- UL64INDGEN

## Non-string Data Type Conversion Routines:

- BYTE
- COMPLEX
- DCOMPLEX
- DOUBLE
- LONG
- LONG64
- UINT
- ULONG

- FIX
- FLOAT

- ULONG64

## Array Manipulation Routines:

- MAX
- MIN
- REPLICATE_INPLACE

- TOTAL
- WHERE

## Programming and IDL Control Routines:

- BYTEORDER
- LOGICAL_AND

- LOGICAL_OR
- LOGICAL_TRUE

# Chapter 11
# Writing Efficient IDL Programs

The following topics are covered in this chapter:

# Overview of Program Efficiency

This chapter presents ideas to consider when trying to create the most efficient programs possible, and discusses how to analyze the performance of your applications.

Knowledge of IDL's implementation and the pitfalls of virtual memory can be used to greatly improve the efficiency of IDL programs. In IDL, complicated computations can be specified at a high level. Therefore, inefficient IDL programs can suffer severe speed penalties — perhaps much more so than with most other languages.

Techniques for writing efficient programs in IDL are identical to those in other computer languages with the addition of the following simple guidelines:

- Use array operations rather than loops wherever possible. Try to avoid loops with high repetition counts. See "Use Vector and Array Operations" on page 196.

- Use IDL system functions and procedures wherever possible. See "Use System Functions and Procedures" on page 199.

- Access array data in machine address order. See "Access Large Arrays by Memory Order" on page 201.

Attention also must be given to algorithm complexity and efficiency, as this is usually the greatest determinant of resources used.

## IDL Implementation

IDL programs are compiled into a low-level abstract machine code which is interpretively executed. The dynamic nature of variables in IDL and the relative complexity of the operators precludes the use of directly executable code. Statements are only compiled once, regardless of the frequency of their execution.

The IDL interpreter emulates a simple stack machine with approximately 50 operation codes. When performing an operation, the interpreter must determine the type and structure of each operand and branch to the appropriate routine. The time required to properly dispatch each operation may be longer than the time required for the operation itself.

The characteristics of the time required for array operations is similar to that of vector computers and array processors. There is an initial set-up time, followed by rapid evaluation of the operation for each element. The time required per element is shorter in longer arrays because the cost of this initial set-up period is spread over more elements. The speed of IDL is comparable to that of optimized FORTRAN for array

operations. When data are treated as scalars, IDL efficiency degrades by a factor of 30 or more.

# Additional Programming Efficiency Resources

Also refer to the following topics, located in other sections of this manual, for additional ways to improve the efficiency of your IDL program:

- "Efficiency and Expression Evaluation Order" on page 245 — describes how to organize operations to increase execution speed

- "Defining and Using Constants" on page 259 — describes the importance of using constants of the correct type

- "Avoid Invariant Expressions" on page 127 — describes the inefficiency of invariant expression within loop statements

# Use Vector and Array Operations

Programs with vector and array expressions run faster than programs with scalars, loops, and IF statements. Whenever possible, vector and array data should be processed with IDL array operations rather than scalar operations in a loop.

## Example—Inverting an Image

Consider the problem of inverting a $512 \times 512$ image. This problem arises because some image display devices consider the origin to be the lower-left corner of the screen, while others recognize it as the upper-left corner.

**Note**
The following example is for demonstration only. The IDL system variable !ORDER should be used to control the origin of image devices. The ORDER keyword to the TV procedure serves the same purpose.

A programmer without experience in using IDL might be tempted to write the following nested loop structure to solve this problem:

```
FOR I = 0, 511 DO FOR J = 0, 255 DO BEGIN

    ;Temporarily save pixel image.
    temp = image[I, J]

    ;Exchange pixel in same column from corresponding row at bottom
    image[I, J] = image[I, 511 - J]

    image[I, 511-J] = temp

ENDFOR
```

A more efficient approach to this problem capitalizes on IDL's ability to process arrays as a single entity:

```
FOR J = 0, 255 DO BEGIN

    ;Temporarily save current row.
    temp = image[*, J]

    ;Exchange row with corresponding row at bottom.
    image[*, J] = image[*, 511-J]

    image[*, 511-J] = temp

ENDFOR
```

At the cost of using twice as much memory, processing can be simplified even further by using the following statements:

```
;Get a second array to hold inverted copy.
image2 = BYTARR(512, 512, /NOZERO)

;Copy the rows from the bottom up.
FOR J = 0, 511 DO image2[*, J] = image[*, 511-J]
```

Even more efficient is the single line:

```
image2 = image[*, 511 - INDGEN(512)]
```

that reverses the array using subscript ranges and array-valued subscripts.

Finally, using the built-in ROTATE function is quickest of all:

```
image = ROTATE(image, 7)
```

Inverting the image is equivalent to transposing it and rotating it 270 degrees clockwise.

See Chapter 15, "Arrays" for complete details on working with arrays in IDL.

## Example—Summing Elements

Consider the problem of adding all positive elements of array B to array A.

Using a loop will be slow:

```
FOR I = 0, (N-1) DO IF B[I] GT 0 THEN A[I] = A[I] + B[I]
```

Masking out negative elements using array operations will be faster:

```
A = A + (B GT 0) * B
```

Adding only the positive elements of B is faster still:

```
A = A + (B > 0)
```

When an IF statement appears in the middle of a loop with each element of an array in the conditional, the loop can often be eliminated by using logical array expressions.

## Example—Using Array Operators and WHERE

In this example, each element of C is set to the square-root of A if A[I] is positive; otherwise, C[I] is set to minus the square-root of the absolute value of A[I].

Using a loop statement is slow:

```
FOR I=0,(N-1) DO IF A[I] LE 0 THEN $
   C[I]=-SQRT(-A[I]) ELSE C[I]=SQRT(A[I])
```

Using an array expression is much faster:

```
C = ((A GT 0) * 2-1) * SQRT(ABS(A))
```

The expression (A GT 0) has the value 1 if A[I] is positive and has the value 0 if A[I]is not. (A GT 0)* 2 - 1 is equal to +1 if A[I] is positive or -1 if A[I] is negative, accomplishing the desired result without resorting to loops or IF statements.

Another method is to use the WHERE function to determine the subscripts of the negative elements of A and negate the corresponding elements of the result.

```
;Get subscripts of negative elements.
negs = WHERE(A LT 0)
;Take root of absolute value.
C = SQRT(ABS(A))
;Negate elements in C corresponding to negative elements in A.
C[negs] = -C[negs]
```

# Use System Functions and Procedures

IDL supplies a number of built-in functions and procedures to perform common operations. These system-supplied functions have been carefully optimized and are almost always much faster than writing the equivalent operation in IDL with loops and subscripting.

## Example

A common operation is to find the sum of the elements in an array or subarray. The TOTAL function directly and efficiently evaluates this sum at least 10 times faster than directly coding the sum.

```
;Slow way: Initialize SUM and sum each element.
sum = 0. & FOR I = J, K DO sum = sum + array[I]

;Efficient, simple way.
sum = TOTAL(array[J:K])
```

Similar savings result when finding the minimum and maximum elements in an array (MIN and MAX functions), sorting (SORT function), finding zero or nonzero elements (WHERE function), etc.

# Virtual Memory

The IDL programmer and user must be cognizant of the characteristics of virtual memory computer systems to avoid penalty. Virtual memory allows the computer to execute programs that require more memory than is actually present in the machine by keeping those portions of programs and data that are not being used on the disk. Although this process is transparent to the user, it greatly affects the efficiency of the program.

**Note** ────────────────────────────────────────────────

In relatively modern computers, plentiful physical memory (hundreds of megabytes for a single-use machine) is not uncommon. Remember, however, that IDL is generally not the only consumer of memory on a system. Other applications, the operating system itself, and other users on multi-user systems may consume large amounts of physical and virtual memory. If your IDL program appears to be inefficient or slow, inspect the system memory situation to determine whether virtual memory is being used, and if so, whether there is enough of it.

────────────────────────────────────────────────────────

IDL arrays are stored in dynamically allocated memory. Although the program can address large amounts of data, only a small portion of that data actually resides in physical memory at any given moment; the remainder is stored on disk. The portion of data and program code in real physical memory is commonly called the working set.

When an attempt is made to access a datum in virtual memory not currently residing in physical memory, the operating system suspends IDL, arranges for the page of memory containing the datum to be moved into physical memory and then allows IDL to continue. This process involves deciding where the datum should go in memory, writing the current contents of the selected memory page out to the disk, and reading the page with the datum into the selected memory page. A *page fault* is said to occur each time this process takes place. Because the time required to read from or write to the disk is very large in relation to the physical memory access time, page faults become an important consideration.

When using IDL with large arrays, it is important to have access to sufficient physical and virtual memory. Given a suitable amount of physical memory, the parameters that regulate virtual memory require adjustment to assure best performance. These parameters are discussed below. See "Virtual Memory System Parameters" on page 203. If you suspect that lack of physical or virtual memory is causing problems, consult your system manager.

# Access Large Arrays by Memory Order

When an array is larger than or close to the working set size (i.e., the amount of physical memory available for the process), it is preferable to access it in memory address order.

Consider the process of transposing a large array. Assume the array is a $512 \times 512$ byte image with a 100 kilobyte working set. The array requires $512 \times 512$, or approximately 250 kilobytes. Less than half of the image can be in memory at any one instant.

In the transpose operation, each row must be interchanged with the corresponding column. The first row, containing the first 512 bytes of the image, will be read into memory, if necessary, and written to the first column. Because arrays are stored in row order (the first subscript varies the fastest), one column of the image spans a range of addresses almost equal to the size of the entire image. To write the first column, 250,000 bytes of data must be read into physical memory, updated, and written back to the disk. This process must be repeated for each column, requiring the entire array be read and written almost 512 times. The amount of time required to transpose the array using the method described above is relatively large.

In contrast, the IDL TRANSPOSE function transposes large arrays by dividing them into subarrays smaller than the working set size enabling it to transpose a $512 \times 512$ image in a much smaller amount of time.

## Example

Consider the operation of the following IDL statement:

```
FOR X = 0, 511 DO FOR Y = 0, 511 DO ARR[X, Y] = ...
```

This statement requires an extremely large execution time because the entire array must be transferred between memory and the disk 512 times. The proper form of the statement is to process the points in address order by using the following statement:

```
FOR Y = 0, 511 DO FOR X = 0, 511 DO ARR[X, Y] = ...
```

This approach cuts computing time by a factor of at least 50.

# Running Out of Virtual Memory

If you process large images with IDL and use the vendor-supplied default system parameters (especially if you have a small system), you may encounter the error message

```
% Unable to allocate memory.
```

This error message means that IDL was unable to obtain enough virtual memory to hold all your data. Whenever you define an array, image, or vector, IDL asks the operating system for some virtual memory in which to store the data. When you reassign the variable, IDL frees the memory for re-use.

The first time you get this error, you will either have to stop what you are doing and exit IDL or delete unused variables containing images or arrays, thereby releasing enough virtual memory to continue. You can delete the memory allocation of array variables by setting the variable equal to a scalar value.

If you need to exit IDL, you first should use the SAVE procedure to save your variables in an IDL save file. Later, you will be able to recover those variables from the save file using the RESTORE procedure.

The HELP,/MEMORY command tells you how much virtual memory you have allocated. For example, a $512 \times 512$ complex floating array requires $8 \times 512^2$ bytes or about 2 megabytes of memory because each complex element requires 8 bytes. Deleting a variable containing a $512 \times 512$ complex array will increase the amount of memory available by this amount.

## Minimizing Virtual Memory

If virtual memory is a problem, try to tailor your programming to minimize the number of images held in IDL variables. Keep in mind that IDL creates temporary arrays to evaluate expressions involving arrays. For example, when evaluating the statement

```
A = (B + C) * (E + F)
```

IDL first evaluates the expression B + C and creates a temporary array if either B or C are arrays. In the same manner, another temporary array is created if either E or F are arrays. Finally, the result is computed, the previous contents of A are deleted, and the temporary area holding the result is saved as variable A. Therefore, during the evaluation of this statement, enough virtual memory to hold two arrays' worth of data is required in addition to normal variable storage.

It is a good idea to delete the allocation of a variable that contains an image and that appears on the left side of an assignment statement, as shown in the following program.

```
;Loop to process an image.
FOR I = ... DO BEGIN

;Processing steps.
...
```

```
;Delete old allocation for A.
A = 0

;Compute image expression and store.
A = Image_Expression

...

;End of loop.
ENDFOR
```

The purpose of the statement A=0 is to free the old memory allocation for the variable A before computing the image expression in the next statement. Because the old value of A is going to be replaced in the next statement, it makes sense to free A's allocation first.

## The TEMPORARY Function

Another way to minimize memory use when performing operations on large arrays is to use the TEMPORARY function. TEMPORARY returns the value of its argument as a temporary variable and makes the argument undefined. In this way, you avoid making a new copy of temporary results. For example, assume that A is a large array. To add 1 to each element in A, you could enter:

```
A = A+1
```

However, this statement creates a new array for the result of the addition and assigns the result to A *before* freeing the old allocation of A. Hence, the total storage required for the operation is twice the size of A. The statement:

```
A = TEMPORARY(A) + 1
```

requires no additional space.

## Virtual Memory System Parameters

The first step is to determine how much virtual memory you require. For example, if you compute complex Fast Fourier Transforms (FFT) on $512 \times 512$ images, each complex image requires 2 megabytes. Suppose that during a typical session you need to have twenty images stored in variables and require enough memory for ten images to hold temporary results, resulting in a total of thirty images or 60 megabytes. Rounding up to 80 megabytes gives a reasonable value for the amount of physical and virtual memory that should be available to IDL.

### UNIX Virtual Memory

For UNIX, The size of the swapping area(s) determines how much virtual memory your process is allowed. To increase the amount of available virtual memory, you must increase the size of the swap device (sometimes called the swap partition). Increasing the size of a swap partition is a time-consuming task that should be planned carefully. It usually requires saving the contents of the disk, reformatting the disk with the new file partition sizes, and restoring the original contents.Some systems offer the alternative of swapping to a regular file. This is a considerably easier solution, although it may not be as efficient. Consult your system documentation for details and instructions on how to perform these operations.

### Windows Virtual Memory

For Microsoft Windows, creation and management of virtual memory files (called "paging files") are handled more or less automatically. You can, however, adjust the initial and maximum size of the paging file for a given disk. Consult your system documentation for details and instructions on how to perform these operations.

# The IDL Code Profiler

The IDL Code Profiler helps you analyze the performance of your applications. You can easily monitor the calling frequency and execution time for procedures and functions. The Profiler can be used with programs entered from the command line as well as programs run from within a file.

You can start the IDL Code Profiler by selecting "Profile" from the Run menu of the IDL Workbench or by entering PROFILER at the Command Line. For more information about the PROFILER procedure, see "PROFILER" (*IDL Reference Guide*).

**Note**
Calling the Profiler from the Command Line does not start the Profiler dialog.

## The Profile Dialog

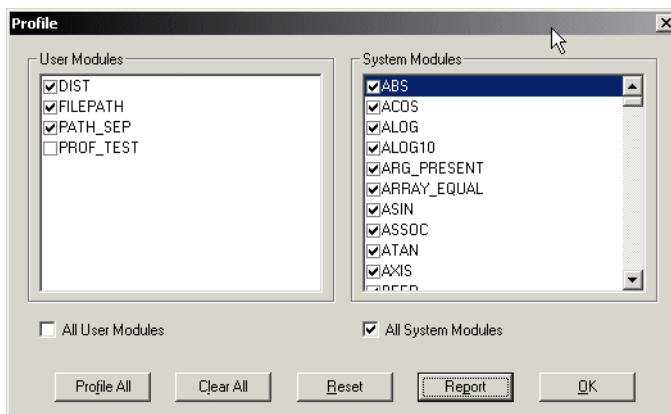Select "Profile" from the Run menu. The Profile dialog appears.



*Figure 11-1: Profile Dialog*

### User Modules

User modules include user-written procedures as well as library procedures and functions provided with IDL. By default, none of the User Modules are selected for

profiling. To select a module, click on the checkbox next to it. All user modules must be compiled before opening the Profile dialog in order to be available for profiling.

### All User Modules

Select this checkbox to select all the user modules for profiling.

## System Modules

This field includes all IDL system procedures and functions.

### All System Modules

Select this checkbox to select all the system modules for profiling.

## Buttons

Click "Profile All" to enable profiling for all the available modules—System and User. Click "Clear All" to disable profiling for all the available modules—System and User. Click "Reset" to clear the report shown in the "Profile Report" dialog. The "Profile Report" dialog is dismissed, as it no longer contains any information. Click "Report" to generate a profile of the selected modules. The Profile Report dialog appears. Click "Cancel" to dismiss the Profile dialog. Click "Help" to display Help on this dialog.

# The Profile Report Dialog

Click "Report" from the Profile dialog in the Run menu of the IDL Workbench. The Profile Report dialog appears.

## Fields in the Profiler Report Dialog

The fields in the Profiler Report dialog show the following attributes of the modules selected for profiling from the Profile dialog. You can sort the values in each column in both ascending and descending order by clicking anywhere within the column. By default, the Modules column is sorted alphabetically.

**Note**
Whether you enter a program at the command line or run a program contained in a file, the PROFILER procedure reports the status of all the modules compiled and executed either since profiling was first set or since the PROFILER was reset.

### Modules

The name of the library, user, or system procedure or function.

**Typ**

The type of module. System procedures or functions are associated with an "S". User or library functions or procedures are associated with a "U".

**Count**

The number of times the procedure or function has been called.

**Only(sec)**

The time required, in seconds, for IDL to execute the given function or procedure, not including any calls to other functions or procedures (children).

**Only Avg**

Average of the Only(sec) field above.

**+Children(sec)**

The time required, in seconds, for IDL to execute the given function or procedure including any calls to other functions or procedures.

**+Child Avg**

Average of the +Children(sec) field above.

## Buttons

Click "Print" to print the report. The Print dialog appears. You can also select "Print" from the File menu of the IDL Workbench. Click "Save" to save the report as a text file. The Save Profile Report dialog appears. Click "Cancel" to dismiss the Profile Report dialog. The contents remain available after cancelling. Click "Help" to display Help on this dialog.

# Using the IDL Code Profiler

Open a new editor file by selecting "New" from the File menu.

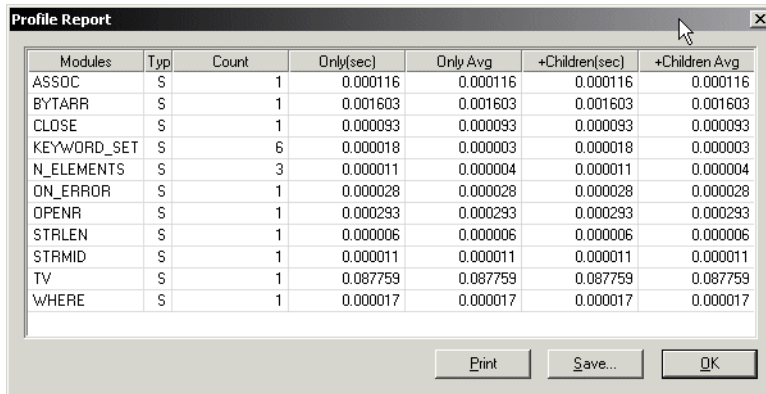Enter the following lines in the editor:

```
PRO prof_test
   OPENR, 1, FILEPATH('nyny.dat', SUBDIR=['examples', 'data'])
   a=ASSOC(1, BYTARR(768,512, /NOZERO))
   b=a[0]
   CLOSE, 1
   TV, b
END
```

Save the file as prof_test.pro by selecting "Save" from the File menu. The Save As dialog appears.

To use the IDL Code Profiler, you must first compile the routines you would like to profile. For more involved programs, you can use RESOLVE_ALL to compile all uncompiled functions or procedures that are called in any already-compiled procedure or function.

Select "Profile..." from the Run menu. The Profile dialog appears; it will remain visible until dismissed. Select "Profile All" to profile all the available modules.

Run the application by selecting "Run" from the File menu. After the application is finished, return to the Profile dialog and click "Report". The Profile Report dialog appears, as shown in the following figure.

| Modules | Typ | Count | Only(sec) | Only Avg | +Children(sec) | +Children Avg |
|---------|-----|-------|-----------|----------|----------------|---------------|
| ASSOC | S | 1 | 0.000116 | 0.000116 | 0.000116 | 0.000116 |
| BYTARR | S | 1 | 0.001603 | 0.001603 | 0.001603 | 0.001603 |
| CLOSE | S | 1 | 0.000093 | 0.000093 | 0.000093 | 0.000093 |
| KEYWORD_SET | S | 6 | 0.000018 | 0.000003 | 0.000018 | 0.000003 |
| N_ELEMENTS | S | 3 | 0.000011 | 0.000004 | 0.000011 | 0.000004 |
| ON_ERROR | S | 1 | 0.000028 | 0.000028 | 0.000028 | 0.000028 |
| OPENR | S | 1 | 0.000293 | 0.000293 | 0.000293 | 0.000293 |
| STRLEN | S | 1 | 0.000006 | 0.000006 | 0.000006 | 0.000006 |
| STRMID | S | 1 | 0.000011 | 0.000011 | 0.000011 | 0.000011 |
| TV | S | 1 | 0.087759 | 0.087759 | 0.087759 | 0.087759 |
| WHERE | S | 1 | 0.000017 | 0.000017 | 0.000017 | 0.000017 |

*Figure 11-2: Profile Report Dialog*

For more information about the capabilities of either dialog, see "The Profile Dialog" on page 205 and "The Profile Report Dialog" on page 206.

## Profiling with Command Line Modules

We will demonstrate how the Profiler handles newly compiled modules. The above example set profiling for all system files, plus the user module, prof_test, and the library function, FILEPATH. If you have altered the above results, reset the report and run prof_test again.

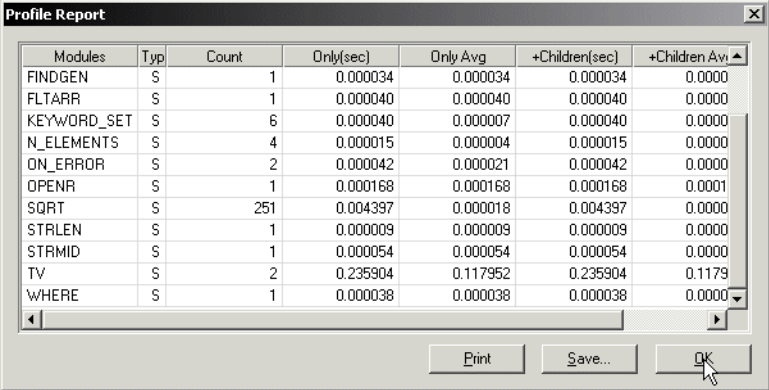Enter the following lines at the Command Line:

```
;Create a dataset using the library function DIST. Note that DIST
;is immediately compiled.
```

```
A= DIST(500)

;Display the image.
TV, A
```

Return to the Profile dialog. You will note that the DIST function has been appended to the User Module field, but that it remains deselected. The Profiler will not include any uncompiled modules by default. Click "Report" in the Profile dialog to refresh the Profile Report dialog's results. The following figure shows the new results. Note that TV is counted twice, and that more system modules have been appended to the Modules column. The DIST function, although it is not itself included, calls system routines which were previously selected for profiling.
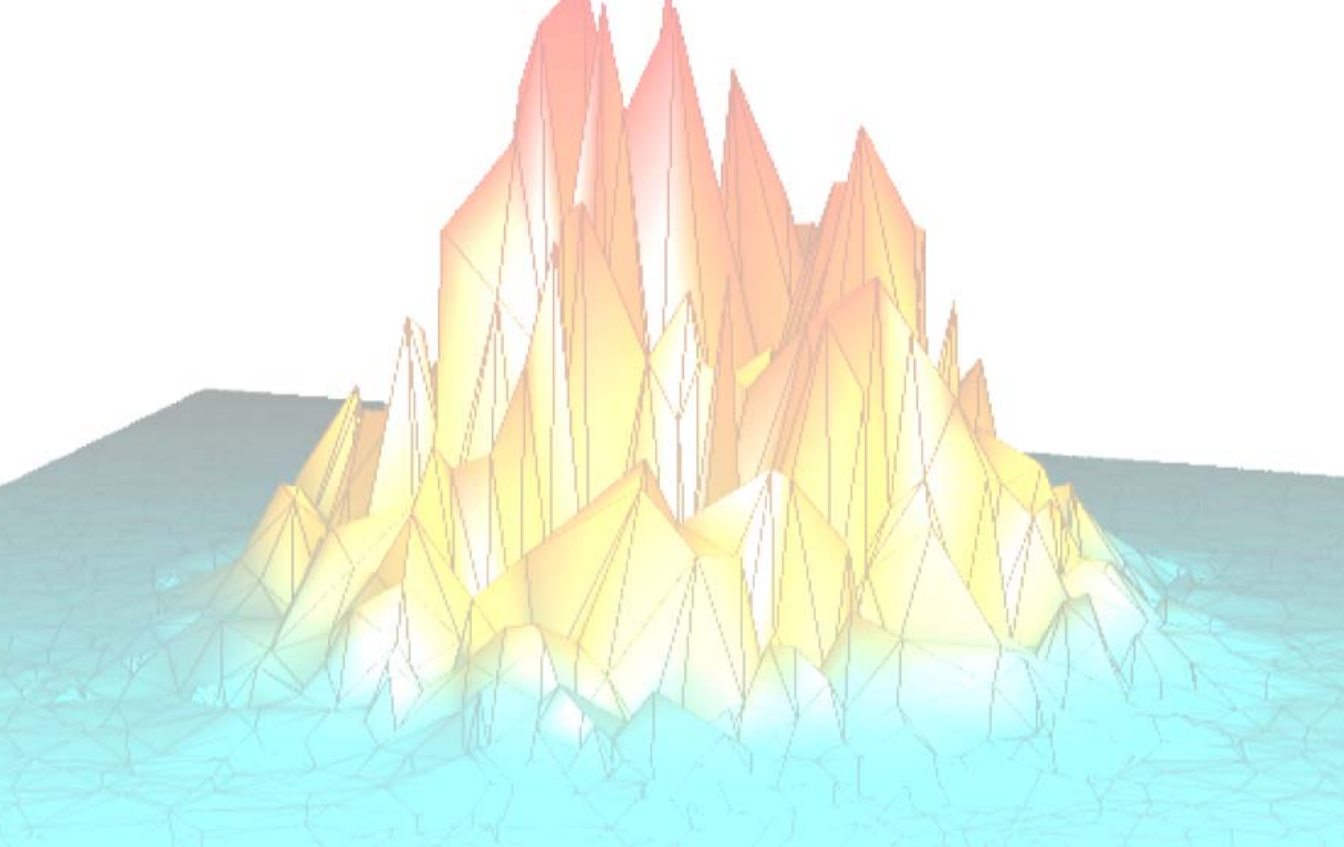


| Modules | Typ | Count | Only(sec) | Only Avg | +Children(sec) | +Children Av |
|---|---|---|---|---|---|---|
| FINDGEN | S | 1 | 0.000034 | 0.000034 | 0.000034 | 0.0000 |
| FLTARR | S | 1 | 0.000040 | 0.000040 | 0.000040 | 0.0000 |
| KEYWORD_SET | S | 6 | 0.000040 | 0.000007 | 0.000040 | 0.0000 |
| N_ELEMENTS | S | 4 | 0.000015 | 0.000004 | 0.000015 | 0.0000 |
| ON_ERROR | S | 2 | 0.000042 | 0.000021 | 0.000042 | 0.0000 |
| OPENR | S | 1 | 0.000168 | 0.000168 | 0.000168 | 0.0001 |
| SQRT | S | 251 | 0.004397 | 0.000018 | 0.004397 | 0.0000 |
| STRLEN | S | 1 | 0.000009 | 0.000009 | 0.000009 | 0.0000 |
| STRMID | S | 1 | 0.000054 | 0.000054 | 0.000054 | 0.0000 |
| TV | S | 2 | 0.235904 | 0.117952 | 0.235904 | 0.1179 |
| WHERE | S | 1 | 0.000038 | 0.000038 | 0.000038 | 0.0000 |

*Figure 11-3: Refreshing the Profile Report*

If you select DIST in the User Modules field in the Profile dialog and then re-enter only the statement calling TV at the Command Line, you will notice that only the count for TV increases in the profiler report. You must re-enter the statement calling DIST at the Command Line; the already-compiled library function is executed again, making it available for profiling.

# Part II: Components of the IDL Language

# Chapter 12
# Expressions and Operators

The following topics are covered in this chapter:

# Overview of Expressions and Operators

Variables, constants, and function results are combined into *expressions* using operators. The value of an expression depends on the values of the *operands* and the operator involved. Expressions can be combined with other expressions, variables, and constants to yield more complex expressions. In IDL, unlike FORTRAN or C, expressions can be scalar- or array-valued.

IDL has a large number of different operators. In addition to the usual operators — addition, subtraction, multiplication, division, exponentiation, relations (EQ, NE, GT, etc.), and logical arithmetic (&&, ||, ~, AND, OR, NOT, and XOR) — other operators exist to find minima, maxima, select scalars and subarrays from arrays (subscripting), and to concatenate scalars and arrays to form new arrays.

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

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

# Mathematical Operators

IDL mathematical operators are described in the following table.

**Note**

Also see "Assignment and Compound Assignment" on page 236 for information on = and op= and "Other Operators" on page 239 for information on the [ ], ( ), and ?: operators.

| Operator | Description | Example |
|:---:|---|---|
| + | Addition | Store the sum of 3 and 6 in B:<br>`B = 3 + 6` |
|  | String Concatenation | Store the string value of "John Doe" in B:<br>`B = 'John' + ' ' + 'Doe'` |
| ++ | Increment | Adds one to the operand:<br>`A = 3`<br>`A++`<br>`PRINT, A`<br><br>IDL Prints:<br>` 4`<br>**Note -** The increment operator supports both pre- and post-fix syntax. See "Using Increment/Decrement" on page 217. |
| – | Subtraction | Store the value of 5 subtracted from 9 in C:<br>`C = 9 – 5` |
|  | Negation | Change the sign of C:<br>`C = -C` |

*Table 12-1: Mathematical Operators*

| Operator | Description | Example |
|:---:|:---|:---|
| -- | Decrement | Subtracts one from the operand:<br><br>```<br>A = 3<br>A--<br>PRINT, A<br>```<br><br>IDL Prints:<br><br>```<br>2<br>```<br>**Note -** The decrement operator supports both pre- and post-fix syntax. See "Using Increment/Decrement" on page 217. |
| * | Multiplication | Store the product of 2 and 5 in variable C:<br><br>```<br>C = 2 * 5<br>``` |
|  | Pointer dereference | If `ptr` is a valid pointer (created via the PTR_NEW function), then `*ptr` is the value held by the heap variable that `ptr` points to. For more information on IDL pointers, see Chapter 17, "Pointers" (*Application Programming*). |
| / | Division | Store result of 10.0 divided by 3.2 in variable D:<br><br>```<br>D = 10.0/3.2<br>``` |

*Table 12-1: Mathematical Operators (Continued)*

| Operator | Description | Example |
|----------|-------------|---------|
| ^ | Exponentiation | Store result of 2 raised to the 3rd power in variable B:<br><br>`B = 2^3`<br>**Note -** How exponentiation is evaluated depends upon whether the operands are real or complex. See "Using Exponentiation" on page 220 for details. |
| MOD | Modulo | I MOD J is equal to the remainder when I is divided by J. The magnitude of the result is less than that of J, and its sign agrees with that of I. Print the value of 9 modulo 5:<br><br>`PRINT, 9 MOD 5`<br><br>IDL Prints:<br><br>`4`<br><br>Compute angle modulo 2p.<br><br>`A =(ANGLE + B) MOD (2 * !PI)` |

*Table 12-1: Mathematical Operators (Continued)*

# Using Increment/Decrement

Beginning in IDL version 6.0, the increment (++) and decrement (--) operators can be applied to variables (including array subscripts or structure tags) of any numeric type. The ++ operator increments the target variable by one. The -- operator decrements the target by one. When written in front of the target variable (that is, using *prefix* notation), the operations are known as *preincrement* and *predecrement*, respectively. When written following the target variable (using *postfix* notation), they are called *postincrement* and *postdecrement*.

**Note**

The increment and decrement operators can only be applied to variable expressions to which a value can be assigned. Hence, the following is not allowed:

```
A = 23++
```

because it attempts to apply the increment operator to a constant. Another way of stating this rule is to say that it must be *possible* for the expression being incremented or decremented to appear on the left-hand side of the equal sign.

The increment and decrement operators can be used either as standalone statements or within a larger enclosing expression. Although the two forms are very similar, the expression form has some efficiency and side-effect issues (described below) that do not apply to the statement form.

## Increment/Decrement Statements

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

- `A++ or ++A`

- `A-- or --A`

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

## Increment/Decrement Expressions

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

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

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

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

### Efficiency of Prefix vs. Postfix Operations

When used as part of an expression, the prefix form of the increment and decrement operators has an efficiency advantage over the postfix form. The reason for this is that the postfix form requires IDL to make a copy of the data, while the prefix form does not. The operations carried out by IDL to execute a prefix increment or decrement operation are:

1. Fetch the target variable.

2. Increment or decrement the target variable in place (no copies are made).

3. Use the variable when evaluating the surrounding expression.

This is very efficient. In contrast, the postfix form requires IDL to make a copy of the variable in order to use its old value in the surrounding expression following the increment/decrement. The operations carried out by IDL to execute a postfix increment or decrement operation are:

1. Fetch the target variable.

2. Make a temporary copy of the variable.

3. Increment or decrement the original variable.

4. Use the temporary copy when evaluating the surrounding expression.

If your computation requires the postfix form, then these operations are necessary and reasonable. If not, the prefix form will use fewer resources and is the better choice. The larger the data involved, the more important this becomes. It is not a concern for small variables.

### Order Of Side Effects

The way that the increment and decrement operators change the value of a variable in addition to using its value in a surrounding expression is called a *side effect*. In most cases, the side effects are desired, and cause no problems. Side effects can cause problems, however, if the increment or decrement operator is applied to a variable that appears more than once within a single statement or expression. Consider the following statement (taken from *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie):

```
A[i] = i++
```

Which value of i is used to index A? Is it the original value of i, or the incremented value? The answer depends on the order in which the various parts of the statement

are evaluated. Either answer might be considered correct, and IDL does not require one or the other. Similarly, in the statements

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

the value of A could be either 47 or 46, depending on which part of the expression is evalutated first.

Note that this situation falls outside the rules of operator precedence — it is the order in which the variables themselves are evalutated that affects the result. Let's examine the situation closely:

- Here the "old" value of B (23) is always used for the first occurrence of B in the statement.

- If the sub-statement B++ is evaluated first, the value of the *second* occurrence of B in the statement uses the "new" value of B (24), giving A the value 47.

- If the sub-statement that contains only the variable B is evaluated first, the "old" value of B will be used for both occurrences, and A will get the value 46.

As with most languages that implement increment and decrement operators, IDL does not require any particular *ordering* of evaluation within an expression in which such side effects occur. Different versions or implementations of IDL may evaluate the same expression differently. As a result, you should avoid writing code that depends on a particular ordering of the side effects.

# Using Exponentiation

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

For real numbers, A^B is evaluated as follows:

- If A is a real number and B is of integer type, repeated multiplication is applied.

- If both A and B are real (non-integer), the formula $A^B = e^{BlnA}$ is evaluated.

- $A^0$ is defined as 1.

For complex numbers, A^B is evalutated as follows. The complex number A can be represented as A = $a + ib$, where $a$ is the real part, and $ib$ is the imaginary part. In polar form, we can represent the complex number as A = $re^{i\theta} = r\cos\theta + ir\sin\theta$, where $r\cos\theta$ is the real part, and $ir\sin\theta$ is the imaginary part:

- If A is complex and B is real, the formula $A^B = (re^{i\theta})^B = r^B(\cos B\theta + i\sin B\theta)$ is evaluated.

- If A is real and B is complex, the formula $A^B = e^{BlnA}$ is evaluated.

- If both A and B are complex, the formula $A^B = e^{BlnA}$ is evaluated, and the natural logarithm is computed to be $\ln(A) = \ln(re^{i\theta}) = \ln(r) + i\theta$.

# Minimum and Maximum Operators

The IDL minimum and maximum operators return the smaller or larger of their operands, as described below.

**Note** —————————————————————————————————
Negated values must be enclosed in parentheses in order for IDL to interpret them correctly.

| Operator | Description | Example |
|:---:|---|---|
| < | Minimum operator. The value of "A < B" is equal to the smaller of A or B.<br><br>**Note -** See also "Using Minimum or Maximum with Complex Numbers" and "Using Minimum or Maximum with NaN Values" below. | Set A equal to 3:<br>`A = 5 < 3`<br><br>Set A equal to -6. Use parentheses to avoid a syntax error.<br>`A = 5 < (-6)`<br><br>Set all points in array ARR that are larger than 100 to 100:<br>`ARR = ARR < 100`<br><br>Set X to the smallest of the three operands:<br>`X = X0 < X1 < X2` |
| > | Maximum operator. "A > B" is equal to the larger of A or B.<br><br>**Note -** See also "Using Minimum or Maximum with Complex Numbers" and "Using Minimum or Maximum with NaN Values" below. | Use '>' to avoid taking the log of zero or negative numbers:<br>`C = ALOG(D > 1E – 6)`<br><br>Plot positive points only. Negative points are plotted as zero:<br>`PLOT, ARR > 0` |

*Table 12-2: Minimum and Maximum Operators*

# Using Minimum or Maximum with Complex Numbers

For complex numbers, the absolute value is used to determine which value is smaller or larger. If both values have the same magnitude then the first value is returned.

## Minimum Operator Examples

```
; Set A equal to 1+2i, since ABS(1+2i) is less than ABS(2-4i):
A = COMPLEX(1,2) < COMPLEX(2,-4)
; Set A equal to 1-2i, since ABS(1-2i) equals ABS(-2+i):
A = COMPLEX(1,-2) < COMPLEX(-2,1)
```

## Maximum Operator Examples

```
; Set A equal to 2-4i, since ABS(2-4i) is greater than ABS(1+2i)
A = COMPLEX(1,2) > COMPLEX(2,-4)

; Set A equal to 1-2i, since ABS(1-2i) equals ABS(-2+i)
A = COMPLEX(1,-2) > COMPLEX(-2,1)
```

# Using Minimum or Maximum with NaN Values

Typically in IDL, the result of any operation involving the special value NaN is simply NaN. For efficiency, IDL does not check the values of A and B for NaN values before performing the minimum or maximum operation. If A or B contains a NaN value, the result is undefined and can be either NaN or the other non-NaN value, depending on the specific hardware and operating system. If you suspect that one of your operands contains NaN values, you might want to use the FINITE function to ensure that you return NaN values in the result. For example, if A and B are scalars:

```
A = !VALUES.F_NAN
B = 5

; Result is undefined and can either be 5 or NaN:
PRINT, A > B

; Result must be NaN if either operand is NaN:
PRINT, ( FINITE(A) && FINITE(B) ) ? ( A > B ) : !VALUES.F_NAN
```

This second method also avoids any floating-point math errors. If A and B are arrays, the following method can be used:

```
C = REPLICATE( !VALUES.F_NAN, N_ELEMENTS(A) )
good = WHERE( FINITE(A) and FINITE(B), ngood )
IF ( ngood GT 0 ) THEN C[good] = A[good] > B[good]
```

# Matrix Operators

IDL has two operators used to multiply arrays and matrices. For an example illustrating the difference between the two, see "Multiplying Arrays" (Chapter 15, *Application Programming*).

| Operator | Description | Example |
|:---:|:---|:---|
| # | Computes array elements by multiplying the columns of the first array by the rows of the second array. The second array must have the same number of columns as the first array has rows. The resulting array has the same number of columns as the first array and the same number of rows as the second array. | Multiply a 3-column by 2-row array:<br><br>```array1 = [ [1, 2, 1], $
    [2, -1, 2] ]```<br><br>Create a 2-column by 3-row array:<br><br>```array2 = [ [1, 3], [0, 1],$
    [1, 1] ]
PRINT, array1#array2```<br><br>IDL prints:<br><br>```7          -1          7
2          -1          2
3           1          3``` |
| ## | Computes array elements by multiplying the rows of the first array by the columns of the second array. The second array must have the same number of rows as the first array has columns. The resulting array has the same number of rows as the first array and the same number of columns as the second array. | Create a 3-column by 2-row array:<br><br>```array1 = [ [1, 2, 1], [2, -1, 2] ]```<br><br>Create a 2-column by 3-row array:<br><br>```array2 = [[1, 3], [0, 1], [1, 1]]
PRINT, array1##array2```<br><br>IDL prints:<br><br>```2          6
4          7``` |

*Table 12-3: Matrix Operators*

**Tip** ——————————————————————————————————————

If one or both of the arrays are also transposed as part of a matrix multiplication, such as TRANSPOSE(A) # B, it is more efficient to use the MATRIX_MULTIPLY function, which does the transpose simultaneously with the multiplication.

———————————————————————————————————————————————

# Logical Operators

Beginning in IDL 6.0, IDL supports three logical operators: &&, ||, and ~. When dealing with logical operators, non-zero numerical values, non-null strings, and non-null heap variables (pointers and object references) are considered true, everything else is false.

**Note** ———————————————————————————————————

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

———————————————————————————————————————————

| Operator | Description | Example |
|:---:|:---|:---|
| && | Logical AND<br><br>Returns 1 whenever both of its operands are true; otherwise, returns 0. Non-zero numerical values, non-null strings, and non-null heap variables (pointers and object references) are considered true, everything else is false.<br><br>Operands must be scalars or single-element arrays. The && operator *short-circuits*; the second operand will not be evaluated if the first is false. See "Short-circuiting" on page 227 for details. | `PRINT, 5 && 7`<br>IDL Prints: 1<br><br>`PRINT, 5 && 2`<br>IDL Prints: 1<br><br>`PRINT, 4 && 0`<br>IDL Prints: 0<br><br>`PRINT, "" && "sun"`<br>IDL Prints: 0 |

*Table 12-4: Logical Operators*

| Operator | Description | Example |
|:---:|:---|:---|
| \|\| | Logical OR<br><br>Returns 1 whenever either of its operands are true; otherwise, returns 0. Uses the same test for "truth" as the && operator.<br><br>Operands must be scalars or single-element arrays. The \|\| operator *short-circuits*; the second operand will not be evaluated if the first is true. See "Short-circuiting" on page 227 for details. | ```IF ((5 GT 3) || (4 GT 5)) $```<br>```    THEN PRINT, 'True'```<br><br>IDL Prints:<br>```True``` |
| ~ | Logical negation<br><br>Returns 1 when its operand is false; otherwise, returns 0. Uses the same test for "truth" as the && operator. | ```PRINT, ~ [1, 2, 0]```<br><br>IDL Prints:<br>```0   0   1``` |

*Table 12-4: Logical Operators (Continued)*

## Short-circuiting

The && and || logical operators, added to IDL in version 6.0, are *short-circuiting* operators. This means that IDL does not evaluate the second operand unless it is necessary in order to determine the proper overall answer. Short-circuiting behavior can be powerful, since it allows you to base the decision to compute the value of the second operand on the value of the first operand. For instance, in the expression:

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

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

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

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

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

# Additional Logical Operator Examples

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

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

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

# Version History

The logical operators &&, ||, and ~, and the LOGICAL_AND and LOGICAL_OR functions were added to IDL in version 6.0.

# Bitwise Operators

There are four bitwise operators in IDL: AND, NOT, OR, and XOR. For integer operands (byte, signed- and unsigned-integer, longword, and 64-bit longword data types), bitwise operators operate on each bit of the operand or operands independently.

| Operator | Description | Example |
|:---:|---|---|
| AND | Bitwise AND<br><br>For integer, longword, and byte operands, a bitwise AND operation is performed. If the operands are scalars, it returns a scalar value. If either operand is an array, it returns an array containing one value for each element of the shortest array operand.<br><br>For operations on other types, the result is equal to the second operand if the first operand is not equal to zero or the null string; otherwise, the result is zero or the null string.<br><br>**Note -** The bitwise AND operator is not valid for heap variable operands | The statement<br>`5 AND 6 = 4`<br>is represented in binary as follows:<br>`0101 AND 0110 = 0100`<br><br>`PRINT, (5 GT 2) AND (4 GT 2)`<br>IDL Prints: 1<br>`PRINT, (5 GT 2) AND (4 GT 5)`<br>IDL Prints: 0<br>`PRINT, 5 AND 7`<br>IDL Prints: 5<br>`PRINT, 5 AND 2`<br>IDL Prints: 0<br>`PRINT, 4 AND 2`<br>IDL Prints: 0 |

*Table 12-5: Logical Operators*

| Operator | Description | Example |
|:---:|:---|:---|
| NOT | Bitwise NOT<br><br>Returns the bitwise inverse of its scalar or array operand (returns scalar if operand is a scalar, or returns an array containing one value for each element of the operand array).<br><br>For integer, longword, and byte operands, NOT returns the complement of each bit of the operand. For floating-point operands, the result is 1.0 if the operand is zero; otherwise, the result is zero.<br><br>**Warning -** Use caution when using the return value from the bitwise NOT operator as an operand for the logical operators `&&` and `||`. See "Using the NOT Operator" on page 232 for additional discussion.<br><br>**Note -** Not valid for string or complex operands. | The statement<br>`NOT 4 = -5`<br>is represented in binary as follows:<br>`NOT 0100 = 1011`<br><br>`PRINT, NOT 1`<br>IDL Prints:<br>`-2`<br><br>**Note -** Modern computers use the "2s complement" representation for negative signed integers. This means that to arrive at the decimal representation of a negative binary number (a string of binary digits with a one as the most significant bit), you must take the complement of each bit, add one, convert to decimal, and prepend a negative sign. For example, NOT 0 equals -1, NOT 1 equals -2, etc.<br><br>`IF (NOT (5 GT 6)) THEN $`<br>`    PRINT, 'True'`<br><br>IDL Prints:<br>`True` |

*Table 12-5: Logical Operators (Continued)*

| Operator | Description | Example |
|----------|-------------|---------|
| OR | Bitwise OR<br><br>Performs the logical "inclusive or" operation on two scalar or array operands (returning a scalar value for scalar operands, or returning an array containing one value for each element of the shortest array operand.<br><br>For integer or byte operands, a bitwise inclusive OR is performed. For floating- point operands, returns the first operand if it is non- zero, or the 2nd operand otherwise. | For integer operands, OR performs a bitwise inclusive "or" operation and returns the result. The statement:<br><br>`3 OR 5 = 7`<br>is represented in binary as follows:<br><br>`0011 OR 0101 = 0111`<br><br>`IF ((5 GT 3) OR $`<br>`   (4 GT 5)) THEN $`<br>`   PRINT, 'True'`<br>IDL Prints:<br><br>`True` |
| XOR | Bitwise exclusive XOR<br><br>XOR is only valid for byte, integer, and longword operands.<br><br>Performs the logical "exclusive or" operation on two scalar or array operands (returning a scalar value for scalar operands, or returning an array containing one value for each element of the shortest array operand.<br><br>A bit in the result is set to 1 if the corresponding bits in the operands are different; if they are equal, it is set to zero. | For integer operands, XOR sets a bit in the result to 1 if the corresponding bits in the operands are different or to 0 if they are equal. The statement:<br><br>`3 XOR 5 = 6`<br>is represented in binary as follows:<br><br>`0011 XOR 0101 = 0110`<br><br>`IF ((5 GT 3) XOR (4  GT 5))`<br>`THEN $`<br>`   PRINT, 'Different' $`<br>`   ELSE PRINT, 'Same'`<br><br>IDL Prints:<br><br>`Different` |

*Table 12-5: Logical Operators (Continued)*

# Using the NOT Operator

Due to the bitwise nature of the NOT operator, logical negation operations should always use ~ in preference to NOT, reserving NOT exclusively for bitwise computations. Consider a statement such as:

```
IF ((NOT EOF(lun)) && device_ready) THEN statement
```

which wants to execute *statement* if the file specified by the variable lun has data remaining, and the variable device_ready is non-zero. When EOF returns the value 1, the expression NOT EOF(lun) yields -2, due to the bitwise nature of the NOT operator. The && operator interprets the value -2 as true, and will therefore attempt to execute *statement* incorrectly in many cases. The proper way to write the above statement is:

```
IF ((~ EOF(lun)) && device_ready) THEN statement
```

# Additional Bitwise Operator Examples

Some examples of bitwise expressions are as follows:

```
; Displays the "negative" of an image contained in the array IMG.
TV, NOT IMG

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

# Relational Operators

The IDL relational operators apply a relation to two operands and return a logical value of true or false. The resulting logical value can be used as the predicate in IF, WHILE or REPEAT statements. You can also combine Boolean operators with other logical values to make more complex expressions.

**Note** ────────────────────────────────────────────────

It is important to see "Definition of True and False" (Chapter 7, *Application Programming*) for details on when a value is considered true or false.

────────────────────────────────────────────────────────

The rules for evaluating relational expressions with operands of mixed modes are the same as for arithmetic expressions. Each operand is promoted to the data type of the operand with the greatest precedence or potential precision. (See "Data Type and Structure of Expressions" on page 252 for details.) For example, in the relational expression "2 EQ 2.0", the integer 2 is converted to floating point and compared to the floating point 2.0. The result of this expression is true. The relational operators return a value of 1 for true and 0 for false. The type of the result is always byte.

**Note** ────────────────────────────────────────────────

When using EQ and NE with complex numbers, both the real and imaginary parts must meet the condition of the relational operator. For example, the following returns 0 (false):

```
PRINT, COMPLEX(1,2) EQ COMPLEX(1,-2)
```

When using GE, GT, LE, and LT with complex numbers, the absolute value (or modulus) of the complex number is used for the comparison.

────────────────────────────────────────────────────────

For more information on using relational operators, also see "Using Relational Operators with Arrays" and "Relational Operators with Infinity and NaN Values" on page 235.

| Operator | Description | Example |
|:---:|:---|:---|
| EQ | Equal to | Returns true if its operands are equal; otherwise, it returns false. The following returns True:<br>`IF (2 EQ 2.0) THEN PRINT, 'True'` |

*Table 12-6: Relational Operators*

| Operator | Description | Example |
|----------|-------------|---------|
| NE | Not equal to | Returns true whenever the operands are different. The following returns 1 (true):<br><br>`PRINT, "sun" NE "fun"` |
| GE | Greater than or equal to | Returns true if the operand on the left is greater than or equal to the one on the right. Relational operator are useful for creating array masks:<br><br>`A = ARRAY * (ARRAY GE 100)`<br><br>See "Using Relational Operators with Arrays" on page 235. |
| GT | Greater than | Returns true if the operand on the left is greater than the operand on the right. Determine if A is greater than B:<br><br>`IF (A GT B) THEN PRINT, 'True'`<br><br>**Note -** Strings are compared using the ASCII collating sequence: " " is less than "0" is less than "9" is less than "A" is less than "Z" is less than "a" which is less than "z". |
| LE | Less than or equal to | Returns true if the operand on the left is less than or equal to the operand on the right. Determine if A is less than or equal to B:<br><br>`IF (A LE B) THEN PRINT, 'True'` |
| LT | Less than | Returns true if the operand on the left is less than the operand on the right. Determine if A is less than B:<br><br>`IF (A LT B) THEN PRINT, 'True'` |

*Table 12-6: Relational Operators (Continued)*

**Note** ———————————————————————————————————

You can use the NE and EQ operators to determine if two object references point to the same heap variable. See "Object Equality and Inequality" (Chapter 1, *Object Programming*) for examples.

———————————————————————————————————————————

# Using Relational Operators with Arrays

Relational operators can be applied to arrays, and the resulting array of ones and zeroes can be used as an operand. For example, the expression:

```
A = ARR * (ARR LE 100)
```

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

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

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

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

# Relational Operators with Infinity and NaN Values

On the Windows platform, using relational operators with the values infinity or NaN (Not a Number) causes an "illegal operand" error. The FINITE function's INFINITY and NAN keywords can be used to perform comparisons involving infinity and NaN values. For more information, see "FINITE" (*IDL Reference Guide*) and "Special Floating-Point Values" on page 158.

# Assignment and Compound Assignment

The assignment statement stores a value in a variable. Compound assignment combines assignment with another operator.

| Operator | Description | Examples |
|:---:|:---|:---|
| = | Assignment<br><br>The value of the expression on the right hand side of the equal sign is stored in the variable, subscript element, or range on the left side. The old value of the variable, if any, is discarded, and the value of the expression is stored in the variable. The expression on the right side can be of any type or structure.<br><br>For more information on assignment involving arrays and ranges, see Chapter 15, "Arrays".<br><br>For information on assignment involving objects, see "Object Assignment" (Chapter 1, *Object Programming*). | Simple assignment examples:<br>`A = 5`<br>Assigns 5 to variable A:<br><br>`B='Hello World'`<br>Assign "Hello World" to variable B:<br>`name = 'Mary'`<br>The variable `name` becomes a scalar string variable.<br>`arr = FLTARR(100)`<br>Make `arr` a 100-element, floating-point array.<br>`arr = arr[50:*]`<br>Discard points 0 to 49 of `arr`. It is now a 50-element array. |

*Table 12-7: Assignment and Compound Assignment*

| Operator | Description | Examples |
|---|---|---|
| *op=* | Compound Assignment where *op* is one of the following operators: ##, #, *, +, -, /, <, >, ^, AND, EQ, GE, GT, LE, LT, MOD, NE, OR, XOR<br><br>Provides succinct syntax for expressions in which the same variable would otherwise be present on both sides of the equal sign.<br><br>See "Compound Assignment Operators" on page 237 for details. | Applies the specified operation to the target variable "in place," without making a copy of the variable. For example,<br><br>`A += 5`<br>adds 5 to the value of the variable A.<br><br>`A op= expression`<br>is equivalent to:<br><br>`A = TEMPORARY(A) op (expression)`<br><br>The following statements both add 100 to current value of A:<br><br>`A = A + 100`<br>`A += 100` |

*Table 12-7: Assignment and Compound Assignment (Continued)*

# Compound Assignment Operators

In addition to the standard assignment statement, IDL versions beginning with 6.0 support the following compound assignment operators:

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

See op= in previous table for examples.

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

```
A op= expression
```

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

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

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

Note that the statement:

```
A op= expression
```

is identical to the IDL statement:

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

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

## Compound Operators and Whitespace

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

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

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

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

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

are identical, but the latter is more readable.

# Other Operators

The following operators (on the [ ], ( ), ?: and -> operators) are used when working with arrays, controlling the order of operations, creating conditional expressions, or invoking an object method.

| Operator | Description | Examples |
|:---:|:---|:---|
| [ ] | Array concatenation<br><br>The expression [A,B] is an array formed by concatenating A and B, which can be scalars or arrays, along the first dimension.<br><br>To concatenate second and third levels, nest the brackets; [[1,2],[3,4]] is a 2-element by 2-element array with the first row containing 1 and 2 and the second row containing 3 and 4. Operands must have compatible dimensions; all dimensions must be equal except the dimension that is to be concatenated, e.g., [2,INTARR(2,2)] are incompatible.<br><br>See Chapter 15, "Arrays" for more information. | Define C as three-point vector:<br>`C = [0, 1, 3]`<br><br>Add 5 to the end of C:<br>`PRINT, [C, 5]`<br>`IDL Prints: 0 1 3 5`<br><br>Insert -1 at the beginning of C:<br>`PRINT, [-1, C]`<br><br>IDL Prints: `-1 0 1 3`<br>Plot ARR2 appended to ARR1.<br>`PLOT, [ARR1, ARR2]`<br><br>Define a 3x3 matrix.<br>`KER = [[1,2,1], [2,4,2], $`<br>`   [1,2,1]]`<br><br>**Note -** Array concatenation is a relatively inefficient operation, and should only be performed once for a given set of data if possible. |
| | Enclose array subscripts<br>**Note -** See "Array Subscript Syntax: [ ] vs. ( )" on page 309 for additional details. | `A = [2, 1, 5]`<br>Print the 3rd element in A:<br>`PRINT, A[2]`<br><br>IDL Prints: 5 |

*Table 12-8: Other Operators*

| Operator | Description | Examples |
|:---:|:---|:---|
| ( ) | Group expressions to control order of evaluation or enclose function parameter lists<br><br>**Note -** See "Operator Precedence" on page 242 for details on order of evaluation | `PRINT, 3 + 4 * 2 ^ 2 /2`<br>IDL Prints: 11<br><br>`PRINT, (3 + (4 * 2) ^ 2 / 2)`<br>IDL Prints: 35<br><br>Enclose function argument lists:<br>`SIN(ANG * PI/180.)` |
| ?: | Conditional expression<br><br>Provides a way to write simple constructions of the IF...THEN...ELSE statement in expression form.<br><br>See "Working with Conditional Expressions" below. | For<br><br>*value = expr1* ? *expr2* : *expr3*<br><br>*expr1* is evaluated first. If *expr1* is true, then *value = expr2*. If *expr1* is false, *value = expr3*.<br><br>`A=6 & B=4`<br><br>Set Z to the greater of A and B:<br><br>`Z = (A GT B) ? A : B`<br>`PRINT, Z`<br><br>IDL Prints: 6 |
| -> | Method invocation<br><br>Calls an object method. See "Acting on Objects Using Methods" (Chapter 1, *Object Programming*) for more information. | `oWindow->Draw`<br>where `oWindow` is an IDLgrWindow object and `Draw` is the object method. |

*Table 12-8: Other Operators (Continued)*

# Working with Conditional Expressions

The conditional expression—written with the ternary operator ?:—has the lowest precedence of all the operators. It provides a way to write simple constructions of the IF...THEN...ELSE statement in expression form. In the following example, Z receives the larger of the values contained by A and B:

```
IF (A GT B) THEN Z = A ELSE Z = B
```

This statement can be written more concisely using a conditional expression:

```
Z = (A GT B) ? A : B
```

The general form of a conditional expression is:

```
expr1 ? expr2 : expr3
```

The expression *expr1* is evaluated first. If *expr1* is true, then the expression *expr2* is evaluated and set as the value of the conditional expression. If *expr1* is false, *expr3* is evaluated and set as the value of the conditional expression. Only one of *expr2* or *expr3* is evaluated, based on the result of *expr1*. (See "Definition of True and False" on page 138 for details on how the "truth" of an expression is determined.)

**Note** ――――――――――――――――――――――――――――――――――――――――

Since ?: has very low precedence—just above assignment—parentheses are not necessary around *expr1*. However, parentheses are often used in this situation, as they enhance the readability of the expression.

――――――――――――――――――――――――――――――――――――――――――――――――

# Operator Precedence

The following table lists IDL's operator precedence. Operators with the highest precedence are evaluated first. Operators with equal precedence are evaluated from left to right.

**Note** ────────────────────────────────────────────────────────
See "Efficiency and Expression Evaluation Order" on page 245 for information on creating efficient statements.
────────────────────────────────────────────────────────────────

| Priority | Operator |
|---|---|
| First (highest) | ( ) (parentheses, to group expressions) |
|  | [ ] (brackets, to concatenate arrays) |
| Second | . (structure field dereference) |
|  | [ ] (brackets, to subscript an array) |
|  | ( ) (parentheses, used in a function call) |
| Third | * (pointer dereference) |
|  | ^ (exponentiation) |
|  | ++ (increment) |
|  | -- (decrement) |
| Fourth | * (multiplication) |
|  | # and ## (matrix multiplication) |
|  | /(division) |
|  | MOD (modulus) |

*Table 12-9: Operator Precedence*

| Priority | Operator |
|---|---|
| Fifth | + (addition) |
| | – (subtraction and negation) |
| | < (minimum) |
| | **>** (maximum) |
| | NOT (bitwise negation) |
| | ~ (logical negation) |
| Sixth | EQ (equality) |
| | NE (not equal) |
| | LE (less than or equal) |
| | LT (less than) |
| | GE (greater than or equal) |
| | GT (greater than) |
| Seventh | AND (bitwise AND) |
| | OR (bitwise OR) |
| | XOR (bitwise exclusive OR) |
| Eighth | && (logical AND) |
| | || (logical OR) |
| Ninth | ?: (conditional expression) |

*Table 12-9: Operator Precedence (Continued)*

**Note** —————————————————————————————————————————————

There is also a data type hierarchy that affects the result of mathematical operations. See "Data Type and Structure of Expressions" on page 252 for details.

———————————————————————————————————————————————————————

The effect of a given operator is based on both position and the rules of operator precedence. This concept is shown by the following examples.

```
A = 4 + 5 * 2
```

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

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

In this case, A equals 18 because the parentheses have higher operator precedence than the multiplication operator; the expression inside the parentheses is evaluated first, and the result is multiplied by two.

Position within the expression is used to determine the order of evaluation when two or more operators share the same operator precedence. Consider the following:

```
A = 6 / 2 * 3
```

In this case, A equals 9, since the division operator is to the left of the multiplication operator. The subexpression 6 / 2 is evaluated before the multiplication is done, even though the multiplication and division operators have the same precedence. Again, parentheses can be used to override the default evaluation order:

```
A = 6 / (2 * 3)
```

In this case, A equals 1, because the expression inside parentheses is evaluated first.

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

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

*Table 12-10: Examples of Expressions*

# Efficiency and Expression Evaluation Order

The order in which an expression is evaluated can have a significant effect on program speed. Consider the following statement, where A is an array:

```
; Scale A from 0 to 16.
B = A * 16. / MAX(A)
```

This statement first multiplies every element in A by 16 and then divides each element by the value of the maximum element. The number of operations required is twice the number of elements in A. A much faster way of computing the same result is used in the following statement:

```
; Scale A from 0 to 16 using only one array operation.
B = A * (16./MAX(A))
```

or

```
; Operators of equal priority are evaluated from left to right.
; Only one array operation is required.
B = 16./MAX(A) * A
```

The faster method only performs one operation for each element in A, plus one scalar division. To see the speed difference on your own machine, execute the following statements:

```
A = RANDOMU(seed, 512, 512)
t1 = SYSTIME(1) & B = A*16./MAX(A) & t2 = SYSTIME(1)
PRINT, 'Time for inefficient calculation: ', t2-t1
t3 = SYSTIME(1) & B = 16./MAX(A)*A & t4 = SYSTIME(1)
PRINT, 'Time for efficient calculation: ', t4-t3
```

# Chapter 13
# Working with Data in IDL

The following topics are covered in this chapter:

# Data Types

The IDL language is *dynamically typed*. This means that an operation on a variable can change that variable's type. In general, when variables of different types are combined in an expression, the result has the data type that yields the highest precision. For example, if an integer variable is added to a floating-point variable, the result will be a floating-point variable. See "Data Type and Structure of Expressions" on page 252

**Note** ───────────────────────────────────────────────────────

See "Returning Type and Size Information" (Chapter 4, *Using IDL*) for information on how to determine the data type of an array.

## Basic Data Types

In IDL there are twelve basic, atomic data types, each with its own form of constant. The data type assigned to a variable is determined either by the syntax used when creating the variable, or as a result of some operation that changes the type of the variable. IDL's basic data types are discussed in more detail beginning with "Defining and Using Constants" on page 259

Table 13-1 lists IDL's basic data types, provides examples of how to explicitly create a variable of each type, and list the routines used to create variables and arrays of each type. Type codes shown in the **Description** column correspond to the type code returned by the SIZE function for that data type.

| Data Type | Description | Creation | Routines |
|-----------|-------------|----------|----------|
| Byte | An 8-bit unsigned integer ranging in value from 0 to 255. Pixels in images are commonly represented as byte data.<br><br>IDL type code: 1 | `a = 5B`<br><br>`a = BYTE(5)` | BYTE<br>BYTARR |

*Table 13-1: Data Types*

| Data Type | Description | Creation | Routines |
|---|---|---|---|
| Integer | A 16-bit signed integer ranging from −32,768 to +32,767.<br><br>IDL type code: 2 | `b = 0`<br><br>`b = 0S`<br><br>`b = FIX(0)` | FIX<br>INTARR |
| Unsigned Integer | A 16-bit unsigned integer ranging from 0 to 65535.<br><br>IDL type code: 12 | `c = 0U`<br><br>`c = UINT(0)` | UINT<br>UINTARR |
| Long | A 32-bit signed integer ranging in value from approximately minus two billion to plus two billion.<br><br>IDL type code: 3 | `d = 0L`<br><br>`d = LONG(0)` | LONG<br>LONARR |
| Unsigned Long | A 32-bit unsigned integer ranging in value from 0 to approximately four billion.<br><br>IDL type code: 13 | `e = 0UL`<br><br>`e = ULONG(0)` | ULONG<br>ULONARR |
| 64-bit Long | A 64-bit signed integer ranging in value from −9,223,372,036,854,775,808 to +9,223,372,036,854,775,807.<br><br>IDL type code: 14 | `f = 0LL`<br><br>`f = LONG64(0)` | LONG64<br>LON64ARR |
| 64-bit Unsigned Long | A 64-bit unsigned integer ranging in value from 0 to 18,446,744,073,709,551,615.<br><br>IDL type code: 15 | `g = 0ULL`<br><br>`g = ULONG64(0)` | ULONG64<br>ULON64ARR |
| Floating-point | A 32-bit, single-precision, floating-point number in the range of $\pm 10^{38}$, with approximately six or seven decimal places of significance.<br><br>IDL type code: 4 | `h = 0.0`<br><br>`h = FLOAT(0)` | FLOAT<br>FLTARR |

*Table 13-1: Data Types (Continued)*

| Data Type | Description | Creation | Routines |
|-----------|-------------|----------|----------|
| Double-precision | A 64-bit, double-precision, floating-point number in the range of $\pm 10^{308}$ with approximately 14 decimal places of significance.<br><br>IDL type code: 5 | `i = 0.0D`<br><br>`i = DOUBLE(0)` | DOUBLE<br>DBLARR |
| Complex | A real-imaginary pair of single-precision, floating-point numbers. Complex numbers are useful for signal processing and frequency domain filtering.<br><br>IDL type code: 6 | `j = $`<br>`COMPLEX(1.0, 0.0)`<br><br>`j = COMPLEX(1,0)` | COMPLEX<br>COMPLEXARR |
| Double-precision complex | A real-imaginary pair of double-precision, floating-point numbers.<br><br>IDL type code: 9 | `k = $`<br>`DCOMPLEX(1.0, 0.0)` | DCOMPLEX<br>DCOMPLEXARR |
| String | A sequence of characters, from 0 to 2147483647 (2.1 GB) characters in length, which is interpreted as text.<br><br>IDL type code: 7 | `l = 'Hello'`<br><br>`l = $`<br>`STRING([72B, 101B, $`<br>`108B, 108B, 111B])` | STRING<br>STRARR |

*Table 13-1: Data Types (Continued)*

**Note**

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

## Precision of Floating-Point Numbers

The precision of IDL's floating-point numbers depends somewhat on the platform involved and the compiler and specific compiler switches used to compile the IDL

executable. The values shown here are minimum values; in some cases, IDL may deliver slightly more precision than we have indicated. If your application uses numbers that are sensitive to floating-point truncation or round-off errors, or values that cannot be represented exactly as floating-point numbers, this is something you should consider.

For more information on floating-point mathematics, see Chapter 9, "Mathematics" (*Using IDL*). For information on your machine's precision, see "MACHAR" (*IDL Reference Guide*).

# Complex Data Types

- Structures: Aggregations of data of various types. Structures are discussed in Chapter 16, "Structures".

- Pointers: A reference to a dynamically-allocated *heap variable*. Pointers are discussed in Chapter 17, "Pointers".

- Object References: A reference to a special heap variable that contains an IDL object structure. Object references are discussed in Chapter 13, "Creating Custom Objects in IDL" (*Object Programming*).

# Data Type and Structure of Expressions

Every entity in IDL has an associated *data type* and *structure*. The *structure of an expression* determines whether the expression can represent a single value or multiple values. IDL expressions can be either *scalars* (with exactly one value) or *arrays* (with one or more values). The data type and structure of an expression depend on the data type and structure of its operands.

**Tip** ────────────────────────────────────────────────

You can determine the data type of an expression by returning the type code of the expression. See "Returning Type and Size Information" (Chapter 4, *Using IDL*) for more information.

────────────────────────────────────────────────

## Hierarchy of IDL Data Types

Unlike many other languages, the data type and structure of most expressions in IDL cannot be determined until the expression is evaluated. Because of this, care must be taken when writing programs. For example, a variable can be a scalar byte variable at one point in a program while at a later point the same variable can hold a complex array. See "Expression Type" on page 253 for information on how the hierarchy of data types affect the outcome of mathematical operations. See "Expression Structure" on page 254 for information on how the results of scalar and array operations are evaluated. The twelve atomic data types in decreasing order of precedence are as follows:

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

# Expression Type

IDL attempts to evaluate expressions containing operands of different data types in the most accurate manner possible. The result of an operation becomes the same data type as the operand with the greatest precedence or potential precision. For example, when adding a byte variable to a floating-point variable, the byte variable is first converted to floating-point, then added to the floating-point variable, yielding a floating-point result. When adding a double-precision variable to a complex variable, the result is double-precision complex, because the double-precision complex type has a higher position in the hierarchy of data types. See "Hierarchy of IDL Data Types" on page 252 for the order of precedence.

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

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

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

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

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

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

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

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

**Note** ─────────────────────────────────────────────────────────────────

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

───────────────────────────────────────────────────────────────────────

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

# Expression Structure

IDL expressions can contain operands that are either scalars or arrays, just as they can contain operands with different types. Conversion of variables between the scalar and array forms is independent of data type conversion. An expression will yield an array result if any of its operands is an array, as shown in the following table:

| Operands | Result |
|----------|--------|
| Scalar : Scalar | Scalar |
| Array : Array | Array |
| Scalar : Array | Array |
| Array : Scalar | Array |

*Table 13-2: Structure of Expressions*

See "Operations on Array Expressions" on page 303 for more information on working with arrays as operands in an expression.

# Date/Time Data

Dates and times are among the many types of information that numerical data can represent. IDL provides a number of routines that offer specialized support for generating, analyzing, and displaying date- and time- based data (herein referred to as date/time data).

## Julian Dates and Times

Within IDL, dates and times are typically stored as Julian dates. A Julian date is defined to be the number of days elapsed since noon on January 1, 4713 BCE. Following the astronomical convention, a Julian day is defined to start at 12pm (noon). The following table shows a few examples of calendar dates and their corresponding Julian dates.

| Calendar Date | Julian Date |
|---|---|
| January 1, 4713 B.C.E., at 12pm | 0 |
| January 2, 4713 B.C.E., at 12pm | 1 |
| January 1, 2000 at 12pm | 2451545 |

*Table 13-3: Example Julian Dates*

Julian dates can also include fractional portions of a day, thereby incorporating hours, minutes, and seconds. If the day fraction is included in a Julian date, it is represented as a double-precision floating point value. The day fraction is computed as follows:

$$dayFraction = \frac{hour}{24.\mathrm{d}} + \frac{minute}{1440.\mathrm{d}} + \frac{seconds}{86400.\mathrm{d}}$$

One advantage of using Julian dates to represent dates and times is that a given date/time can be stored within a single variable (rather than storing the year, month, day, hour, minute, and second information in six different variables). Because each Julian date is simply a number, IDL's numerical routines can be applied to Julian dates just as for any other type of number.

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

## **Precision of Date/Time Data**

The precision of any numerical value is defined as the smallest possible number that
can be added to that value that produces a new value different from the first.
Precision is typically limited by the data type of the variable used to store the number
and the magnitude of the number itself. Within IDL, the following guide should be
used when choosing a data format for date/time data:

- Time values that require a high precision, and that span a range of a few days
  or less, should be stored as double-precision values in units of "time elapsed"
  since the starting time, rather than in Julian date format. An example would be
  the "seconds elapsed" since the beginning of an experiment. In this case, the
  data can be treated within IDL as standard numeric data without the need to
  utilize IDL's specialized date/time features.

- Date values that do not include the time of day may be stored as long-integer
  Julian dates. The Julian date format has the advantage of being compact (one
  value per date) and being evenly spaced in days. As an example, January 1st
  for the years 2000, 2001, and 2002 can be stored as Julian days 2451545,
  2451911, and 2452276. The precision of this format is 1 day.

- Date values where it is necessary to include the time of day can be stored as
  double-precision Julian dates, with the time included as a day fraction.
  Because of the large magnitude of the Julian date (such as Julian day 2451545
  for 1 January 2000), the precision of most Julian dates is limited to 1
  millisecond (0.001 seconds).

To determine the precision of a Julian date/time value, you can use the IDL
MACHAR function:

```
; Set date to January 1, 2000, at 12:15pm:
julian = JULDAY(1,1,2000,12,15,0)

; Get machine characteristics:
machine = MACHAR(/DOUBLE)

; Multiply by floating-point precision:
precision = julian*machine.eps

; Convert to seconds:
PRINT, precision*86400d0
```

# How to Generate Date/Time Data

The TIMEGEN function returns an array of double precision floating point values that represent date/time in terms of Julian dates. The first value of the returned array corresponds to a start date/time, and each subsequent value corresponds to the start date/time plus that array element's one-dimensional subscript multiplied by a step size for a given date/time unit. Unlike the other array generation routines in IDL, TIMEGEN includes a START keyword, which is necessary if the starting date/time is originally provided in calendar (month, day, year) form.

The following example begins with a start date of March 1, 2000 and increments every month for a full year:

```
date_time = TIMEGEN(12, UNIT = 'Months', $
    START = JULDAY(3, 1, 2000))
```

where the UNIT keyword is set to 'Months' to increment by month and the START keyword is set to the Julian date form of March 1, 2000. The results of the above call to TIMEGEN can be output using either of the following methods:

1.  Using the CALDAT procedure to convert the Julian dates to calendar dates:

```
CALDAT, date_time, month, day, year
FOR i = 0, (N_ELEMENTS(date_time) - 1) DO PRINT, $
    month[i], day[i], year[i], $
    FORMAT = '(i2.2, "/", i2.2, "/", i4)'
```

2.  Using the calendar format codes:

```
PRINT, date_time, format = '(C(CMOI2.2, "/", CDI2.2, "/", CYI))'
```

The resulting calendar dates are printed out as follows:

```
03/01/2000
04/01/2000
05/01/2000
06/01/2000
07/01/2000
08/01/2000
09/01/2000
10/01/2000
11/01/2000
12/01/2000
01/01/2001
02/01/2001
```

The TIMEGEN routine contains several keywords to provide specific date/time data generation. For more information, see the "TIMEGEN" (*IDL Reference Guide*).

# Date/Time Data Examples

You can display date/time data on IDLgrAxis objects (through the TICKFORMAT property) plots, contours, and surfaces by setting tick mark attributes. See "Displaying Date/Time Data on Axis Objects" (Chapter 5, *Object Programming*) and the routines LABEL_DATE and "CONTOUR" (*IDL Reference Guide*) routine for examples.

# Defining and Using Constants

The syntax of a constant determines its type. Efficiency is adversely affected when the type of a constant must be converted during expression evaluation. Consider the following expression:

```
A + 5
```

If the variable A is of floating-point type, the constant 5 must be converted from short integer type to floating point each time the expression is evaluated.

The type of a constant also has an important effect in array expressions. Care must be taken to write constants of the correct type. In particular, when performing arithmetic on byte arrays with the intent of obtaining byte results, be sure to use byte constants; e.g., *n*B. For example, if A is a byte array, the result of the expression A + 5B is a byte array, while A + 5 yields a 16-bit integer array.

This section discusses details of IDL data types including the following:

- "Integer Constants" below
- "Floating-Point and Double-Precision Constants" on page 262
- "Complex Constants" on page 264
- "String Constants" on page 264

# Integer Constants

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

| Radix | Type | Form | Examples |
|---|---|---|---|
| Decimal | Byte | nB | 12B, 34B |
| | Integer | n or nS | 12,12S,425,425S |
| | Unsigned Integer | nU or nUS | 12U,12US |
| | Long | nL | 12L, 94L |
| | Unsigned Long | nUL | 12UL, 94UL |
| | 64-bit Long | nLL | 12LL, 94LL |
| | Unsigned 64-bit Long | nULL | 12ULL, 94ULL |
| Hexadecimal | Byte | 'n'XB | '2E'XB |
| | Integer | 'n'X or 'n'XS | '0F'X, 'A2'XS |
| | Unsigned Integer | 'n'XU or 'n'XUS | '0F'XU, 'A2'XUS |
| | Long | 'n'XL | 'FF'XL |
| | Unsigned Long | 'n'XUL | 'FF'XUL |
| | 64-bit Integer | 'n'XLL | 'FF'XLL |
| | Unsigned 64-bit Integer | 'n'XULL | 'FF'XULL |

*Table 13-4: Integer Constants*

| Radix | Type | Form | Examples |
|-------|------|------|----------|
| Octal | Byte | "nB | "12B |
| | Integer | "n | "12 |
| | | 'n'O or 'n'OS | '377'O, '234'OS |
| | Unsigned Integer | "nU | "12U |
| | | 'n'OU or 'n'OUS | '377'OU, '234'OUS |
| | Long | "nL | "12L |
| | | 'n'OL | '777777'OL |
| | Unsigned Long | "nUL | "12UL |
| | | 'n'OUL | '777777'OUL |
| | 64-bit Long | "nLL | "12LL |
| | | 'n'OLL | '777777'OLL |
| | Unsigned 64-bit | "nULL | "12ULL |
| | Long | 'n'OULL | '777777'OULL |

*Table 13-4: Integer Constants (Continued)*

Digits in hexadecimal constants include the letters A through F for the decimal numbers 10 through 15. Octal constant use the same style as hexadecimal constants, substituting an O for the X. Absolute values of integer constants are given in the following table.

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

*Table 13-5: Absolute Value Range Of Integer Constants*

| Type | Absolute Value Range |
|------|---------------------|
| 64-bit Long | $0 - 2^{63} - 1$ |
| Unsigned 64-bit Long | $0 - 2^{64} - 1$ |

*Table 13-5: Absolute Value Range Of Integer Constants (Continued)*

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

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

*Table 13-6: Examples of Integer Constants*

# Floating-Point and Double-Precision Constants

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

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

| Form | Example |
|------|---------|
| *n.* | 102. |
| *.n* | .102 |
| *n.n* | 10.2 |
| *n*E | 10E |
| *n*E*sx* | 10E5 |
| *n.*E*sx* | 10.E-3 |
| *.n*E*sx* | .1E+12 |
| *n.n*E*sx* | 2.3E12 |

*Table 13-7: Syntax of Floating-Point Constants*

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

**Note** ───────────────────────────────────────────────

The nE and nD forms are shorthand for nE0 and nD0, and are usually used to indicate the *type* of the number, either single or double precision. When using these forms in expressions, be sure to leave a space after the E or D if the next term has a + or - sign.

For example, the expression 1D+45 is evaluated as $1 \times 10^{45}$ in double precision, while 1D + 45 (note the spaces) evaluates to the number 46 in double precision. Similarly, the expression 1D+x gives an error, because there was no space after the D. The correct way to write this expression is 1D + x (note the spaces).

───────────────────────────────────────────────────────

# Complex Constants

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

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

or

```
COMPLEX(REAL_PART)
```

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

# String Constants

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

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

*Table 13-8: Examples of Valid String Constants*

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

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

*Table 13-9: Examples of Invalid String Constants*

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

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

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

**Note** ─────────────────────────────────────────────────────────

See Chapter 14, "Strings" for details on working with strings.

─────────────────────────────────────────────────────────────────

# Accuracy and Floating Point Operations

In a computer, real numbers are represented with finite precision. While in most cases it is safe to assume that the result of an arithmetical operation done on your computer is correct, it is important to remember that this finite-precision representation leads to unavoidable errors, especially when floating-point numbers, which are digital approximations to real numbers, are involved.

To understand why floating-point numbers are inherently inaccurate, consider the following:

- Floating-point numbers must be made to fit in a space (a string of binary digits in a computer's memory register) that can only hold an integer and a scaling factor.

- Floating-point numbers are represented by strings of a limited number of bits, but represent numbers much larger or smaller than that number of digits can be made to express.

In other words, floating-point values are finite-precision approximations of infinitely precise numbers.

## Roundoff Error

When working with floating-point arithmetic, it is helpful to consider the quantity known as the machine accuracy or the floating-point accuracy of your particular computer. This is the smallest number that, when added to 1.0, produces a floating-point result that is different from 1.0.

A useful way of thinking about machine accuracy is to consider it to be the fractional accuracy to which floating-point numbers are represented. In other words, the machine accuracy roughly corresponds to a change of the least significant bit of the floating-point mantissa—precisely what can happen if a number with more significant digits than fit in the floating-point mantissa is rounded to fit the space available. Generally speaking, every floating-point arithmetic operation introduces an error at least equal to the machine accuracy into the result. This error is known as roundoff error.

Roundoff errors are cumulative. Depending on the algorithm you are using, a calculation involving $n$ arithmetic operations might have a total roundoff error between SQRT($n$) times the machine accuracy and $n$ times the machine accuracy.

Note that the machine accuracy is not the same as the smallest floating-point number your computer can represent. To find these and other machine-dependent quantities for your own computer, see MACHAR in the *IDL Reference Guide*.

# Truncation Error

Another type of error is also present in some numerical algorithms. Truncation error is the error introduced by the process of numerically approximating a continuous function by evaluating it at a finite number of discrete points. Often, accuracy can be increased (again at some cost of computation time) by increasing the number of discrete points evaluated.

For example, consider the process of calculating

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!}$$

Obviously, the answer becomes more accurate as *n* approaches infinity. When performing the actual computation, however, a cutoff value must be specified for *n*. Increasing *n* reduces truncation error at the expense of computational effort.

Several IDL routines allow you to specify cutoff values in such cases (see, for example, INT_2D of the *IDL Reference Guide*). When writing your own routines in IDL, it is important to consider this trade-off between accuracy and computational time.

# Routines for Mathematical Error Assessment

Below is a brief description of IDL routines for checking math error status and machine characteristics. More detailed information is available in the *IDL Reference Guide*.

| | |
|---|---|
| CHECK_MATH | Returns and clears accumulated math error status. |
| FINITE | Returns True if its argument is finite. |
| MACHAR | Determines and returns machine-specific parameters affecting floating-point arithmetic. |

*Table 13-10: Mathematical Error Assessment Routines in IDL*

See "Math Errors" on page 157 for more information.

## Accuracy and Floating Point Operation References

Burden, Richard L., J. Douglas Faires, and Albert C. Reynolds. *Numerical Analysis*. Boston: PWS Publishing, 1993. ISBN 0-534-93219-3

Stoer, J., and R. Bulirsch. *Introduction to Numerical Analysis*. New York: Springer-Verlag, 1980. ISBN 0-387-90420-4

Press, William H. et al. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge: Cambridge University Press, 1992. ISBN 0-521-43108-5

# Type Conversion Functions

IDL allows you to convert data from one data type to another using a set of
conversion functions. These functions are useful when you need to force the
evaluation of an expression to a certain type, output data in a mode compatible with
other programs, etc. For a list of type conversion functions, see "Type Conversion"
(*IDL Quick Reference*). Conversion functions operate on data of any structure:
scalars, vectors, or arrays, and variables can be of any type.

## Take Care When Converting Types

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

```
; Define A. Note that the value of A is outside the range
; of integers, and is automatically created as a longword
; integer by IDL.
A = 33000
;B is silently truncated.
B = FIX(A)
PRINT, B
```

IDL prints:

```
-32536
```

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

With most floating-point operations, error conditions can be monitored using the
FINITE and CHECK_MATH functions. See "Math Errors" on page 157, for more
information.

## Converting Strings

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

Conversion between strings and byte arrays (or vice versa) is something of a special
case. The result of the BYTE function applied to a string or string array is a byte array
containing the ASCII codes of the characters of the string. Converting a byte array

with the STRING function yields a string array or scalar with one less dimension than the byte array.

## Dynamic Type Conversion

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

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

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

    ; Prompt the user for a numeric value:
    READ, UserVal, PROMPT='Enter any Numeric Value: '
    ; Convert the user value to the type stored in typeA:
    ConvUserVal = FIX(UserVal, TYPE=typeA)

    PRINT, ConvUserVal
END
```

## Examples of Type Conversion

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

| Operation | Results |
|---|---|
| FLOAT(1) | 1.0 |
| FIX(1.3 + 1.7) | 3 |
| FIX(1.3) + FIX(1.7) | 2 |
| FIX(1.3, TYPE=5) | 1.3000000 |

*Table 13-11: Uses of Type Conversion Functions*

| Operation | Results |
|---|---|
| BYTE(1.2) | 1 |
| BYTE(-1) | 255b (Bytes are modulo 256) |
| BYTE('01ABC') | [48b, 49b, 65b, 66b, 67b] |
| STRING([65B, 66B, 67B]) | 'ABC' |
| FLOAT(COMPLEX(1, 2)) | 1.0 |
| COMPLEX([1, 2], [4, 5]) | [COMPLEX(1,4),COMPLEX(2,5)] |

*Table 13-11: Uses of Type Conversion Functions (Continued)*

# Variables

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

## Attributes of Variables

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

### Structure

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

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

### Type

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

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

```
ABC = DEF
```

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

# Variable Names

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

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

*Table 13-12: Unacceptable and Acceptable IDL Variable Names*

**Tip** ————————————————————————————

Use the IDL_VALIDNAME routine to determine whether a given string is acceptable as an IDL variable name.

**Warning** ————————————————————————

A variable cannot have the same name as a function (either built-in or user-defined) or a reserved word (see "Reserved Words" (*IDL Reference Guide*)). Giving a variable such a name results in a syntax error or in "hiding" the variable.

# System Variables

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

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

System variables are discussed in Appendix D, "System Variables" (*IDL Reference Guide*).

# Chapter 14
# Strings

The following topics are covered in this chapter:

# Overview of Strings

An IDL string is a sequence of characters from 0 to 2147483647 (2.1 GB) characters in length. Strings have dynamic length (they grow or shrink to fit), and there is no need to declare the maximum length of a string prior to using it. As with any data type, string arrays can be created to hold more than a single string. In this case, the length of each individual string in the array depends only on its own length and is not affected by the lengths of the other string elements.

**Note** ──────────────────────────────────────────────────────────
This chapter covers operations on strings. For information about using the ' and "characters to create valid strings, see "String Constants" on page 264.
───────────────────────────────────────────────────────────────────

## A Note About the Examples

In some of the examples in this chapter, it is assumed that a string array named TREES exists. TREES contains the names of seven trees, one name per element, and is created using the statement:

```
trees = ['Beech', 'Birch', 'Mahogany', 'Maple', 'Oak', $
         'Pine', 'Walnut']
```

Executing the statement,

```
PRINT, '>' + trees + '< '
```

results in the following output:

```
>Beech<  >Birch<  >Mahogany<  >Maple<  >Oak<  >Pine<  >Walnut<
```

# String Operations

IDL supports several basic string operations, as described below:

| Operation | Description |
|---|---|
| Concatenation | The Addition operator, "+", can be used to concatenate strings together. See "String Concatenation" on page 279. |
| Formatting Data | The STRING function is used to format data into a string. The READS procedure can be used to read values from a string into IDL variables. See "Using STRING to Format Data" on page 280. |
| Case Folding | The STRLOWCASE function returns a copy of its string argument converted to lowercase. Similarly, the STRUPCASE function converts its argument to uppercase. See "Case Folding" on page 284. |
| White Space Removal | The STRCOMPRESS and STRTRIM functions can be used to eliminate unwanted white space (blanks or tabs) from their string arguments. See "Whitespace" on page 285. |
| Length | The STRLEN function returns the length of its string argument. See "Finding the Length of a String" on page 287. |
| Substrings | The STRPOS, STRPUT, and STRMID routines locate, insert, and extract substrings from their string arguments. See "Substrings" on page 288. |
| Splitting and Joining Strings | The STRSPLIT function is used to break strings apart, and the STRJOIN function can be used to and glue strings together. See "Splitting and Joining Strings" on page 291 |
| Comparing Strings | The STRCMP, STRMATCH, and STREGEX functions perform string comparisons. See "Comparing Strings" on page 292. |

*Table 14-1: String Operations*

# Non-string and Non-scalar Arguments

Most of the string processing routines described in this chapter expect at least one argument — the string on which they act. If the argument is not of type string, IDL converts it to type string using the same default formatting rules that are used by the PRINT/PRINTF or STRING routines. The function then operates on the converted result. Thus, the IDL statement,

```
PRINT, STRLEN(23)
```

returns the result

```
8
```

because the argument "23" is first converted to the string '      23' that happens to be a string of length 8.

If the argument is an array instead of a scalar, the function returns an array result with the same structure as the argument. Each element of the result corresponds to an element of the argument. For example, the following statements:

```
; Create array of trees.
trees = ['Beech', 'Birch', 'Mahogany', 'Maple', 'Oak', $
        'Pine', 'Walnut']

; Get an uppercase version of TREES.
A = STRUPCASE(trees)

; Show that the result is also an array.
HELP, A

; Display the original.
PRINT, trees

; Display the result.
PRINT, A
```

produce the following output:

```
A               STRING    = Array(7)
Beech Birch Mahogany Maple Oak Pine Walnut
BEECH BIRCH MAHOGANY MAPLE OAK PINE WALNUT
```

For more details on how individual routines handle their arguments, see the individual descriptions in the *IDL Reference Guide.*

# String Concatenation

The addition operator is used to concatenate strings. For example, the command:

```
A = 'This is' + ' a concatenation example.'
PRINT, A
```

results in the following output:

```
This is a concatenation example.
```

Strings can also be broken across code lines:

```
Print, "This is a multi-line " $
   + "string concatenation example."
```

results in the following output:

```
This is a multiline string concatenation example.
```

The following IDL statements build a scalar string containing a comma-separated list of the names found in the TREES string array:

```
; Create array of trees.
trees = ['Beech', 'Birch', 'Mahogany', 'Maple', 'Oak', $
         'Pine', 'Walnut']

; Use REPLICATE to make an array with the correct number of commas
; and add it to trees.
names = trees + [REPLICATE(',', N_ELEMENTS(trees)-1), '']

; Show the resulting list.
PRINT, names
```

Running the above statements results in the following output:

```
Beech, Birch, Mahogany, Maple, Oak, Pine, Walnut
```

# Using STRING to Format Data

The STRING function has the following form:

S = STRING(*Expression$_1$*, ..., *Expression$_n$*)

It converts its parameters to characters, returning the result as a string expression. It is identical in function to the PRINT procedure, except that its output is placed into a string rather than being output to the terminal. As with PRINT, the FORMAT keyword can be used to explicitly specify the desired format. See the discussions of free format and explicitly formatted input/output ("Free Format I/O" on page 387) for details of data formatting. For more information on the STRING function, see "STRING" (*IDL Reference Guide*).

As a simple example, the following IDL statements:

```
; Produce a string array.
A = STRING(FORMAT='("The values are:", /, (I))', INDGEN(5))

; Show its structure.
HELP, A

; Print the result.
FOR I = 0, (N_ELEMENTS(A)-1) DO PRINT, A[I]
```

produce the following output:

```
A    STRING    = Array(6)
The values are:
     0
     1
     2
     3
     4
```

**Note** ────────────────────────────────────────────────────

When you use vector, TrueType, and some device fonts, text strings can include embedded formatting commands that facilitate subscripting, superscripting, and equation formatting. See "Embedded Formatting Commands" (Appendix H, *IDL Reference Guide*).

────────────────────────────────────────────────────────────

# Reading Data from Strings

The READS procedure performs formatted input from a string variable and writes the results into one or more output variables. This procedure differs from the READ procedure only in that the input comes from memory instead of a file.

This routine is useful when you need to examine the format of a data file before reading the information it contains. Each line of the file can be read into a string using READF. Then the components of that line can be read into variables using READS.

See the description of "READS" (*IDL Reference Guide*) for more details.

# Byte Arguments and Strings

There is a close association between a string and a byte array—a string is simply an array of bytes that is treated as a series of ASCII characters. Therefore, it is convenient to be able to convert between them easily.

When STRING is called with a single argument of byte type and the FORMAT keyword is not used, STRING does not work in its normal fashion. Instead of formatting the byte data and placing it into a string, it returns a string containing the byte values from the original argument. Thus, the result has one less dimension than the original argument. A two-dimensional byte array becomes a vector of strings, and a byte vector becomes a scalar string. However, a byte scalar also becomes a string scalar. For example, the statement

```
PRINT, STRING([72B, 101B, 108B, 108B, 111B])
```

produces the output below:

```
Hello
```

This output results because the argument to STRING, as produced by the array concatenation operator, is a byte vector. Its first element is 72B which is the ASCII code for "H," the second is 101B which is an ASCII "e," and so forth. The PRINT keyword can be used to disable this feature and cause STRING to treat byte data in the usual way.

As discussed in Chapter 18, "Files and Input/Output", it is easier to read fixed-length string data from binary files into byte variables instead of string variables. Therefore, it is convenient to read the data into a byte array and use this special behavior of STRING to convert the data into string form.

Another use for this feature is to build strings that contain nonprintable characters in a way such that the character is not entered directly. This results in programs that are easier to read and that also avoid file transfer difficulties (some forms of file transfer have problems transferring nonprintable characters). Due to the way in which strings are implemented in IDL, applying the STRING function to a byte array containing a null (zero) value will result in the resulting string being truncated at that position. Thus, the statement,

```
PRINT, STRING([65B, 66B, 0B, 67B])
```

produces the following output:

```
AB
```

This output is produced because the null byte in the third position of the byte array argument terminates the string and hides the last character.

**Note** ───────────────────────────────────────────────

The BYTE function, when called with a single argument of type string, performs the inverse operation to that described above, resulting in a byte array containing the same byte values as its string argument. For additional information about the BYTE function, see "Type Conversion Functions" on page 269.

───────────────────────────────────────────────

# Case Folding

The STRLOWCASE and STRUPCASE functions are used to convert arguments to lowercase or uppercase. Where *String* is the string to be converted, they have the form:

S = STRLOWCASE(*String*)

S = STRUPCASE(*String*)

The following IDL statements generate a table of the contents of TREES showing each name in its actual case, lowercase and uppercase:

```
; Create array of trees.
trees = ['Beech', 'Birch', 'Mahogany', 'Maple', 'Oak', $
        'Pine', 'Walnut']

FOR I=0, 6 DO PRINT, trees[I], STRLOWCASE(trees[I]),$
STRUPCASE(trees[I]), FORMAT = '(A, T15, A, T30, A)'
```

The resulting output from running this statement is as follows:

| | | |
|---|---|---|
| Beech | beech | BEECH |
| Birch | birch | BIRCH |
| Mahogany | mahogany | MAHOGANY |
| Maple | maple | MAPLE |
| Oak | oak | OAK |
| Pine | pine | PINE |
| Walnut | walnut | WALNUT |

A common use for case folding occurs when writing IDL procedures that require input from the user. By folding the case of the response, it is possible to handle responses written in uppercase, lowercase, or mixed case. For example, the following IDL statements can be used to ask "yes or no" style questions:

```
; Create a string variable to hold the response.
answer = ''

; Ask the question.
READ, 'Answer yes or no:  ', answer
IF (STRUPCASE(answer) EQ 'YES') THEN $
   ; Compare the response to the expected answer.
   PRINT,'YES' ELSE PRINT, 'NO'
```

# Whitespace

The STRCOMPRESS and STRTRIM functions are used to remove unwanted white space (tabs and spaces) from a string. This can be useful when reading string data from arbitrarily formatted strings.

## Removing All Whitespace

The function STRCOMPRESS returns a copy of its string argument with all white space replaced with a single space or completely removed. It has the form:

S = STRCOMPRESS(*String*)

where *String* is the string to be compressed.

The default action is to replace each section of white space with a single space. Setting the REMOVE_ALL keyword causes white space to be completely eliminated. For example,

```
; Create a string with undesirable white space. Such a string might
; be the result of reading user input with a READ statement.
A = '   This   is a poorly   spaced   sentence.   '

; Print the result of shrinking all white space to a single blank.
PRINT, '>', STRCOMPRESS(A), '<'

; Print the result of removing all white space.
PRINT '>', STRCOMPRESS(A, /REMOVE_ALL), '<'
```

results in the output:

```
> This is a poorly spaced sentence. <
>Thisisapoorlyspacedsentence.<
```

## Removing Leading or Trailing Blanks

The function STRTRIM returns a copy of its string argument with leading and/or trailing white space removed. It has the form:

S = STRTRIM(*String*[, *Flag*])

where *String* is the string to be trimmed and *Flag* is an integer that indicates the specific trimming to be done. If *Flag* is 0 or is not present, trailing white space is removed. If it is 1, leading white space is removed. Both trailing and leading white space are removed if *Flag* is equal to 2. For example:

```
; Create a string with unwanted leading and trailing blanks.
```

```
A = ' This string has leading and trailing white space   '

; Remove trailing white space.
PRINT, '>', STRTRIM(A), '<'

; Remove leading white space.
PRINT, '>', STRTRIM(A,1), '<'

; Remove both.
PRINT, '>', STRTRIM(A,2), '<'
```

Executing these statements produces the output below.

```
>     This string has leading and trailing white space<
>This string has leading and trailing white space    <
>This string has leading and trailing white space<
```

# Removing All Types of Whitespace

When processing string data, STRCOMPRESS and STRTRIM can be combined to remove leading and trailing white space and shrink any white space in the middle down to single spaces.

```
; Create a string with undesirable white space.
A = 'Yet   another  poorly spaced   sentence.   '

; Eliminate unwanted white space.
PRINT, '>' STRCOMPRESS(STRTRIM(A,2)),   '<'
```

Executing these statements gives the result below:

```
>Yet another poorly spaced sentence.<
```

# Finding the Length of a String

The STRLEN function is used to obtain the length of a string. It has the form:

L = STRLEN(*String*)

where *String* is the string for which the length is required. For example, the following statement

```
PRINT, STRLEN('This sentence has 31 characters')
```

results in the output

```
31
```

while the following IDL statement prints the lengths of all the names contained in the array TREES.

```
; Create array of trees.
trees = ['Beech', 'Birch', 'Mahogany', 'Maple', 'Oak', $
         'Pine', 'Walnut']

PRINT, STRLEN(trees)
```

The resulting output is as follows:

```
       5           5           8           5           3           4           6
```

# Substrings

IDL provides the STRPOS, STRPUT, and STRMID routines to locate, insert, and extract substrings from their string arguments.

## Searching for a Substring

The STRPOS function is used to search for the first occurrence of a substring. It has the form

    S = STRPOS(*Object*, *Search_string*[, *Position*])

where *Object* is the string to be searched, *Search_string* is the substring to search for, and *Position* is the character position (starting with position 0) at which the search is begun. If the optional argument *Position* is omitted, the search is started at the first character (character position 0). The following IDL procedure counts the number of times that the word "dog" appears in the string "dog cat duck rabbit dog cat dog":

```
PRO Animals

; The search string, "dog", appears three times.
animals = 'dog cat duck rabbit dog cat dog'

; Start searching in character position 0.
I = 0

; Number of occurrences found.
cnt = 0

; Search for an occurrence.
WHILE (I NE -1) DO BEGIN
   I = STRPOS(animals, 'dog', I)

   IF (I NE -1) THEN BEGIN
     ; Update counter.
     cnt = cnt + 1

     ;I ncrement I so as not to count the same instance of 'dog'
     ; twice.
     I = I + 1

   ENDIF
ENDWHILE

; Print the result.
PRINT, 'Found ', cnt, " occurrences of 'dog'"
END
```

Running the above program produces the result below.

```
Found          3 occurrences of 'dog'
```

# Searching For the Last Occurrence of a Substring

The REVERSE_SEARCH keyword to the STRPOS function makes it easy to find the last occurrence of a substring within a string. In the following example, we search for the last occurrence of the letter "I" (or "i") in a sentence:

```
sentence = 'IDL is fun.'
sentence = STRUPCASE(sentence)
lasti = STRPOS(sentence, 'I', /REVERSE_SEARCH)
PRINT, lasti
```

This results in:

```
4
```

Note that although REVERSE_SEARCH tells STRPOS to begin searching from the end of the string, the STRPOS function still returns the position of the search string starting from the beginning of the string (where 0 is the position of the first character).

# Inserting the Contents of One String into Another

The STRPUT procedure is used to insert the contents of one string into another. It has the form,

STRPUT, *Destination*, *Source*[, *Position*]

where *Destination* is the string to be overwritten, *Source* is the string to be inserted, and *Position* is the first character position within *Destination* at which *Source* will be inserted. If the optional argument *Position* is omitted, the overwrite is started at the first character (character position 0). The following IDL statements use STRPOS and STRPUT to replace every occurrence of the word "dog" with the word "CAT" in the string "dog cat duck rabbit dog cat dog":

```
animals = 'dog cat duck rabbit dog cat dog'
;The string to search, "dog", appears three times.

;While any occurrence of "dog" exists, replace it.
WHILE (((I = STRPOS(animals, 'dog'))) NE -1) DO $
STRPUT, animals, 'CAT', I

;Show the resulting string.
PRINT, animals
```

Running the above statements produces the result below.

```
CAT cat duck rabbit CAT cat CAT
```

# Extracting Substrings

The STRMID function is used for extracting substrings from a larger string. It has the form:

   STRMID(*Expression*, *First_Character* [, *Length*])

where *Expression* is the string from which the substring will be extracted, *First_Character* is the starting position within *Expression* of the substring (the first position is position 0), and *Length* is the length of the substring to extract. If there are not *Length* characters following the position *First_Character,* the substring will be truncated. If the *Length* argument is not supplied, STRMID extracts all characters from the specified starting position to the end of the string. The following IDL statements use STRMID to print a table matching the number of each month with its three-letter abbreviation:

```
; String containing all the month names.
months = 'JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC'

; Extract each name in turn. The equation (I-1)*3 calculates the
; position within MONTH for each abbreviation
FOR I = 1, 12 DO PRINT, I, '       ', $
STRMID(months, (I - 1) * 3, 3)
```

The result of executing these statements is as follows:

```
 1      JAN
 2      FEB
 3      MAR
 4      APR
 5      MAY
 6      JUN
 7      JUL
 8      AUG
 9      SEP
10      OCT
11      NOV
12      DEC
```

# Splitting and Joining Strings

The STRSPLIT function is used to break apart a string, and the STRJOIN function is used to glue together separate strings into a single string.

The STRSPLIT function uses the following syntax:

*Result* = STRSPLIT( *String* [, *Pattern*] )

where *String* is the string to be split, and *Pattern* is either a string of character codes used to specify the delimiter, or a regular expression, as implemented by the STREGEX function.

The STRJOIN function uses the following syntax:

*Result* = STRJOIN( *String* [, *Delimiter*] )

where *String* is the string or string array to be joined, and *Delimiter* is the separator string to use between the joined strings.

The following example uses STRSPLIT to extract words from a sentence into an array, modifies the array, and uses STRJOIN to rejoin the individual array elements into a new sentence:

```
str1 = 'Hello Cruel World'
words = STRSPLIT(str1, ' ', /EXTRACT)
newwords=[words[0],words[2]]
PRINT, STRJOIN(newwords, ' ')
```

This code results in the following output:

```
Hello World
```

In this example, the EXTRACT keyword caused STRSPLIT to return the substrings as array elements, rather than the default action of returning an array of character offsets indicating the position of each substring.

The STRJOIN function allows us to specify the delimiter used to join the strings. Instead of using a space as in the above example, we could use a different delimiter as follows:

```
str1 = 'Hello Cruel World'
words = STRSPLIT(str1, ' ', /EXTRACT)
newwords=[words[0],words[2]]
PRINT, STRJOIN(newwords, ' Kind ')
```

This code results in the following output:

```
Hello Kind World
```

# Comparing Strings

IDL provides several different mechanisms for performing string comparisons. In addition to the EQ operator, the STRCMP, STRMATCH, and STREGEX functions can all be used for string comparisons.

## Case-Insensitive Comparisons of the First N Characters

The STRCMP function simplifies case-insensitive comparisons, and comparisons of only the first N characters of two strings. The STRCMP function uses the following syntax:

> *Result* = STRCMP( *String1*, *String2* [, *N*] )

where *String1* and *String2* are the strings to be compared, and *N* is the number of characters from the beginning of the string to compare.

Using the EQ operator to compare the first 3 characters of the strings "Moose" and "mOO" requires the following steps:

```
A = 'Moose'
B = 'mOO'

C=STRMID(A,0,3)

IF (STRLOWCASE(C) EQ STRLOWCASE(B)) THEN PRINT, "It's a match!"
```

Using the EQ operator for this case-insensitive comparison of the first 3 characters requires the STRMID function to extract the first 3 characters, and the STRLOWCASE (or STRUPCASE) function to change the case.

The STRCMP function could be used to simplify this comparison:

```
A='Moose'
B='mOO'

IF (STRCMP(A,B,3, /FOLD_CASE) EQ 1) THEN PRINT, "It's a match!"
```

The optional *N* argument of the STRCMP function allows us to easily specify how many characters to compare (from the beginning of the input strings), and the FOLD_CASE keyword specifies a case-insensitive search. If *N* is omitted, the full strings are compared.

# String Comparisons Using Wildcards

The STRMATCH function can be used to compare a search string containing wildcard characters to another string. It is similar in function to the way the standard UNIX command shell processes file wildcard characters.

The STRMATCH function uses the following syntax:

*Result* = STRMATCH( *String*, *SearchString* )

where *String* is the string in which to search for *SearchString*.

*SearchString* can contain the following wildcard characters:

| Wildcard Character | Description |
|---|---|
| * | Matches any string, including the null string. |
| ? | Matches any single character. |
| [...] | Matches any one of the enclosed characters. A pair of characters separated by "-" matches any character lexically between the pair, inclusive. If the first character following the opening [ is a !, any character not enclosed is matched. To prevent one of these characters from acting as a wildcard, it can be quoted by preceding it with a backslash character (e.g. "\*" matches the asterisk character). Quoting any other character (including \ itself) is equivalent to the character (e.g. "\a" is the same as "a"). |

*Table 14-2: Wildcard Characters used by STRMATCH*

The following examples demonstrate various uses of wildcard matching:

**Example 1:** Find all 4-letter words in a string array that begin with "f" or "F" and end with "t" or "T":

```
str = ['foot', 'Feet', 'fate', 'FAST', 'ferret', 'fort']
PRINT, str[WHERE(STRMATCH(str, 'f??t', /FOLD_CASE) EQ 1)]
```

This results in:

```
foot Feet FAST fort
```

**Example 2:** Find words of any length that begin with "f" and end with "t":

```
str = ['foot', 'Feet', 'fate', 'FAST', 'ferret', 'fort']
```

```
PRINT, str[WHERE(STRMATCH(str, 'f*t', /FOLD_CASE) EQ 1)]
```

This results in:

```
foot Feet FAST ferret fort
```

**Example 3:** Find 4-letter words beginning with "f" and ending with "t", with any combination of "o" and "e" in between:

```
str = ['foot', 'Feet', 'fate', 'FAST', 'ferret', 'fort']
PRINT, str[WHERE(STRMATCH(str, 'f[eo][eo]t', /FOLD_CASE) EQ 1)]
```

This results in:

```
foot Feet
```

**Example 4:** Find all words beginning with "f" and ending with "t" whose second character is not the letter "o":

```
str = ['foot', 'Feet', 'fate', 'FAST', 'ferret', 'fort']
PRINT, str[WHERE(STRMATCH(str, 'f[!o]*t', /FOLD_CASE) EQ 1)]
```

This results in:

```
Feet FAST ferret
```

# Complex Comparisons Using Regular Expressions

A more difficult search than the one above would be to find words of any length beginning with "f" and ending with "t" without the letter "o" in between. This would be difficult to accomplish with STRMATCH, but could be easily accomplished using the STREGEX function:

```
str = ['foot', 'Feet', 'fate', 'FAST', 'ferret', 'fort']
PRINT, STREGEX(str, '^f[^o]*t$', /EXTRACT, /FOLD_CASE)
```

This statement results in:

```
Feet FAST ferret
```

Note the following about this example:

• Unlike the * wildcard character used by STRMATCH, the * meta character used by STREGEX applies to the item directly on its left, which in this case is [^o], meaning "any character except the letter 'o' ". Therefore, [^o]* means "zero or more characters that are not 'o' ", whereas the following statement would find only words whose second character is not "o":

```
PRINT, str[WHERE(STRMATCH(str, 'f[!o]*t', /FOLD_CASE) EQ 1)]
```

- The anchors (^ and $) tell STREGEX to find only words that begin with "f" and end with "t". If we left out the $ anchor, STREGEX would also return "fat", which is a substring of "fate".

Regular expressions are somewhat more difficult to use than simple wildcard matching (which is why the UNIX shell does matching) but in exchange offers unparalleled expressive power.

For more on the STREGEX function, see "STREGEX" (*IDL Reference Guide*), and for an introduction to regular expressions, see "Learning About Regular Expressions" on page 297.

# Non-Printing Characters

ASCII characters with value less than 32 or greater than 126 do not have printable representations. Such characters can be included in string constants by specifying their ASCII value as a byte argument to the STRING function.

For example, to represent the TAB character, use the expression

```
STRING(9B)
```

This syntax can be used when comparing strings or performing regular expression matching. For example, to find the position of the first TAB character in a string:

```
pos = STREGEX(input_string, STRING(9b))
```

where *input_string* is a variable containing the string to be searched.

The following table lists the some ASCII characters you might commonly want to represent as IDL strings.

| ASCII Character | Byte Value |
|---|---|
| Bell | 7B |
| Backspace | 8B |
| Horizontal Tab | 9B |
| Linefeed | 10B |
| Vertical Tab | 11B |
| Formfeed | 12B |
| Carriage Return | 13B |
| Escape | 27B |

*Table 14-3: Selected ASCII Characters
and Their Byte Values*

For a complete list, consult a standard ASCII table.

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

# Learning About Regular Expressions

Regular expressions are a very powerful way to match arbitrary text. Stemming from neurophysiological research conducted in the early 1940's, their mathematical foundation was established during the 1950's and 1960's. Their use has a long history in computer science, and they are an integral part of many UNIX tools, including awk, egrep, lex, perl, and sed, as well as many text editors. Regular expressions are slower than simple pattern matching algorithms, and they can be cryptic and difficult to write correctly. Small mistakes in specification can yield surprising results. They are, however, vastly more succinct and powerful than simple pattern matching, and can easily handle tasks that would be difficult or impossible otherwise.

The topic of regular expressions is a very large one, complicated by the arbitrary differences in the implementations found in various tools. Anything beyond an extremely simplistic sketch is well beyond the scope of this manual. To understand them better, we recommend a good text on the subject, such as "Mastering Regular Expressions", by Jeffrey E.F. Friedl (O'Reilly & Associates, Inc, ISBN 1-56592-257-3). The following is an abbreviated, simplified, and incomplete explanation of regular expressions, sufficient to gain a cursory understanding of them.

The regular expression engine attempts to match the regular expression against the input string. Such matching starts at the beginning of the string and moves from left to right. The matching is considered to be "greedy", because at any given point, it will always match the longest possible substring. For example, if a regular expression could match the substring 'aa' or 'aaa', it will always take the longer option.

## Meta Characters

A regular expression "ordinary character" is a character that matches itself. Most characters are ordinary. The exceptions, sometimes called "meta characters", have special meanings. To convert a meta character into an ordinary one, you "escape" it by preceding it with a backslash character (e.g. '\*').

The meta characters are described in the following table:

| Character | Description |
|---|---|
| . | The period matches any character. |
| [ ] | The open bracket character indicates a "bracket expression", which is discussed below. The close bracket character terminates such an expression. |
| \ | The backslash suppresses the special meaning of the character it precedes, and turns it into an ordinary character. To insert a backslash into your regular expression pattern, use a double backslash ('\\'). |
| ( ) | The open parenthesis indicates a "subexpression", discussed below. The close parenthesis character terminates such a subexpression. |
| **Repetition Characters** | These characters below are used to specify repetition. The repetition is applied to the character or expression directly to the left of the repetition operator. |
| * | Zero or more of the character or expression to the left. Hence, 'a*' means "zero or more instances of 'a' ". |
| + | One or more of the character or expression to the left. Hence, 'a+' means "one or more instances of 'a'". |
| ? | Zero or one of the character or expression to the left. Hence, 'a?' will match 'a' or the empty string ''. |
| { } | An interval qualifier allows you to specify exactly how many instances of the character or expression to the left to match. If it encloses a single unsigned integer length, it means to match exactly that number of instances. Hence, 'a{3}' will match 'aaa'. If it encloses 2 such integers separated by a comma, it specifies a range of possible repetitions. For example, 'a{2,4}' will match 'aa', 'aaa', or 'aaaa'. Note that '{0,1}' is equivalent to '?'. |

*Table 14-4: Meta Characters*

| Character | Description |
|---|---|
| \| | Alternation. This operator is used to indicate that one of several possible choices can match. For example, '(a\|b\|c)z' will match any of 'az', 'bz', or 'cz'. |
| ^ $ | Anchors. A '^' matches the beginning of a string, and '$' matches the end. As we have seen above, regular expressions usually match any possible substring. Anchors can be used to change this and require a match to occur at the beginning or end of the string. For example, '^abc' will only match strings that start with the string 'abc'. '^abc$' will only match a string containing *only* 'abc'. |

*Table 14-4: Meta Characters (Continued)*

# Subexpressions

Subexpressions are those parts of a regular expression enclosed in parentheses. There are two reasons to use subexpressions:

- To apply a repetition operator to more than one character. For example, '(fun){3}' matches 'funfunfun', while 'fun{3}' matches 'funnn'.

- To allow location of the subexpression using the SUBEXPR keyword to STREGEX.

# Bracket Expressions

Bracket expressions (expressions enclosed in square brackets) are used to specify a set of characters that can satisfy a match. Many of the meta characters described above (.*[\) lose their special meaning within a bracket expression. The right bracket loses its special meaning if it occurs as the first character in the expression (after an initial '^', if any).

There are several different forms of bracket expressions, including:

- **Matching List** — A matching list expression specifies a list that matches any one of the characters in the list. For example, '[abc]' matches any of the characters 'a', 'b', or 'c'.

- **Non-Matching List** — A non-matching list expression begins with a '^', and specifies a list that matches any character *not* in the list. For example, '[^abc]' matches any characters *except* 'a', 'b', or 'c'. The '^' only has this special meaning when it occurs first in the list immediately after the opening '['.

- • **Range Expression** — A range expression consists of 2 characters separated by a hyphen, and matches any characters lexically within the range indicated. For example, '[A-Za-z]' will match any alphabetic character, upper or lower case. Another way to get this effect is to specify '[a-z]' and use the FOLD_CASE keyword to STREGEX.

# Special Characters in Regular Expressions

Special (non-printing) characters are often represented in regular expressions using backslash escape codes, such as \t to represent a TAB character or \n to represent a newline character. IDL does not support these backslash codes in regular expressions. See "Non-Printing Characters" on page 296 for information on how to represent these special characters in regular expressions.

# Chapter 15
# Arrays

The following topics are covered in this chapter:

# Overview of Arrays

*Arrays* are multidimensional data sets which are manipulated according to mathematical rules. An array can be of any IDL data type; saying that an array is of a particular type means that all elements of the array are of that data type. Array *subscripts* provide a means of selecting one or more elements of an array for retrieval or modification.

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

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

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

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

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

```
PRINT, array
```

IDL prints:

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

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

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

IDL is an array-oriented language. This means that any operation on an array is performed on all elements of the array, without the need for the user to write an explicit loop. The resulting code is easier to read and understand, and executes more efficiently. For example, suppose you have a three-dimensional array and wish to divide each element by two. A language that does not support array operations would require you to write a loop to perform the division for each element; IDL can accomplish the division in a single line of code:

```
array = array/2
```

# Determining the Number of Array Elements

The N_ELEMENTS function returns the number of elements contained in any expression or variable. Scalars always have one element. The number of elements in arrays or vectors is equal to the product of the dimensions. The N_ELEMENTS function returns zero if its parameter is an undefined variable. The result is always a longword scalar. For example, the following expression is equal to the mean of a numeric vector or array.

```
array = FINDGEN(3, 4, 5)
PRINT, TOTAL(array) / N_ELEMENTS(array)
```

# Operations on Array Expressions

Functions exist to create arrays of the data types IDL supports. (See "Array Creation" (*IDL Quick Reference*) for a list of available routines.) The dimensions of the desired array are the parameters to these functions. The result of FLTARR(5) is a floating-point array with one dimension, a vector, with five elements initialized to zero. FLTARR(50,100) is a two-dimensional array, a matrix, with 50 columns and 100 rows.

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

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

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

```
; An array in which each element is equal to the same element in
; ARR plus one. The result has the same dimensions as ARR. If ARR
; is byte or integer, the result is of integer type; otherwise, the
; result is the same type as ARR.
ARR + 1

; An array obtained by summing two arrays.
ARR1 + ARR2
```

```
; An array in which each element is set to twice the smaller of
; either the corresponding element of ARR or 100.
(ARR < 100) * 2

; An array in which each element is equal to the exponential of the
; same element of ARR divided by 10.
EXP(ARR/10.)

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

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

# Array Subscripts

*Subscripts* are used to select individual elements of an array for retrieval or modification. The subscript of an array element denotes the address of the element within the array. In the simple case of a one-dimensional array (that is, an *n*-element vector), elements are numbered starting at 0 with the first element, 1 for the second element, and running to $n-1$, the subscript of the last element.

The syntax of a subscript reference is:

> *Variable_Name [Subscript_ List]*

or

> *(Array_Expression)[Subscript_List]*

The *Subscript_List* is simply a list of expressions, constants, or subscript ranges containing the values of one or more subscripts. Subscript expressions are separated by commas if there is more than one subscript. In addition, multiple elements are selected with subscript expressions that contain either a contiguous range of subscripts or an array of subscripts. Factors affecting the outcome of the expression include whether the subscript appears on the right or left side of the assignment operator, and the dimensionality of the subscript (scalar, array or range). See the following topics for more information:

- See "Understanding Array Subscripts" on page 306 for important information regarding the structure of an array and how subscripts are used to access elements of the array

- See "Assignment Operations and Arrays" on page 310 for details on how to manipulate arrays using subscripts and the assignment operator

- See "Manipulating Arrays" on page 326 for information on transposing and multiplying multi-dimensional arrays

- "Columns, Rows, and Array Majority" on page 332 describes how a multi-dimensional array is mapped in computer memory, and the ramifications of this mapping when working with arrays in IDL

# Understanding Array Subscripts

Subscripts can be used either to retrieve the value of one or more array elements or to designate array elements to receive new values. The expression `arr[12]` denotes the value of the 13th element of `arr` (because subscripts start at 0), while the statement `arr[12] = 5` stores the number 5 in the 13th element of `arr` without changing the other elements.

Elements of multidimensional arrays are specified by using one subscript for each dimension. IDL's notational convention is that for generic arrays and images, the first subscript denotes the column and the second subscript denotes the row. In standard mathematical representation (linear algebra, for example), the convention is reversed: the first subscript denotes the row and the second subscript denotes the column.

If `A` is a 2-element by 3-element array (using [*column*, *row*] notation), the elements are stored in memory as follows:

|         |         | **Stored in Memory** |
|---------|---------|----------------------|
| $A_{0,0}$ | $A_{1,0}$ | Lowest memory address |
| $A_{0,1}$ | $A_{1,1}$ | .<br>.<br>. |
| $A_{0,2}$ | $A_{1,2}$ | Highest memory address |

*Table 15-1: Storage of IDL Array Elements in Memory*

The elements are ordered in memory as: $A_{0,0}$, $A_{1,0}$, $A_{0,1}$, $A_{1,1}$, $A_{0,2}$, $A_{1,2}$. This ordering is like Fortran. It is the opposite of the order used by C/C++. For more information on how IDL arranges multidimensional data in memory, see "Columns, Rows, and Array Majority" on page 332. For a discussion of how the ordering of such data relates to IDL mathematics routines, see "Manipulating Arrays" on page 326.

**Note** ──────────────────────────────────────

When comparing IDL's memory layout to other languages, remember that those
languages usually use a mathematical [*row, column*] notation for array dimensions,
which is the reverse of the array notation used for the example above. See
"Columns, Rows, and Array Majority" on page 332 for more on comparing IDL's
array layout to that of other languages.

────────────────────────────────────────────────

Arrays that contain image data are usually displayed in graphics displays with row
zero at the bottom of the screen, matching the display's coordinate system (although
this order can be reversed by setting the system variable !ORDER to a nonzero
value). Array data are printed to standard text output (such as the IDL output log or
console window) with the first row on top.

Arrays with multiple dimensions are addressed by specifying a subscript expression
for each dimension. A two-dimensional array with *n* columns and *m* rows, is
addressed with a subscript of the form *[i, j]*, where $0 \le i < n$ and $0 \le j < m$. The first
subscript, *i*, is the column index; the second subscript, *j*, is the row index. For
example, the following statements select and print the element in the first column of
the second row of array:

```
array = [[1, 2, 3], [4, 5, 6]]
PRINT, array[0,1]
```

IDL prints:

```
4
```

Elements of multidimensional arrays also can be specified using only one subscript,
in which case the array is treated as a vector with the same number of points.

$A_{0,0}$     $A_{0,1}$

$A_{0,1}$     $A_{1,1}$

$A_{0,2}$     $A_{1,2}$

In the 2 by 3 element array, A, element A[2] is the same element as A[0, 1], and
A[5] is the same element as A[1, 2].

If an attempt is made to reference a nonexistent element of an array using a scalar
subscript (a subscript that is negative or larger than the size of the dimension
minus 1), an error occurs and program execution stops.

Subscripts can be any type of vector or scalar expression. If a subscript expression is
not integer, a longword integer copy is made and used to evaluate the subscript.

**Note**

    When floating-point numbers are converted to longword integers, they are
    truncated, not rounded. Thus, specifying A[1.99] is the same as specifying A[1].

# Extra Dimensions

When creating arrays, IDL eliminates all size 1, or "degenerate", trailing dimensions.
Thus, the statements

```
A = INTARR(10, 1)
HELP, A
```

print the following:

```
A               INT       = Array[10]
```

This removal of superfluous dimensions is usually convenient, but it can cause
problems when attempting to write fully general procedures and functions. Therefore,
IDL allows you to specify "extra" dimensions for an array as long as the extra
dimensions are all zero.

For example, consider a vector defined as follows:

```
arr = INDGEN(10)
```

The following are all valid references to the sixth element of arr:

```
X = arr[5]
X = arr[5, 0]
X = arr[5, 0, 0, *, 0]
```

Thus, the automatic removal of degenerate trailing dimensions does not cause
problems for routines that attempt to access the resulting array.

The REFORM function can be used to add degenerate trailing dimensions to an array
if desired. For example, the following statements create a 10 element integer vector,
and then alter the dimensions to be [10, 1]:

```
A = INTARR(10)
A = REFORM(A, 10, 1, /OVERWRITE)
```

# Array Subscript Syntax: [ ] vs. ( )

Versions of IDL prior to version 5.0 used parentheses to indicate array subscripts. Function calls use parentheses in a visually identical way to specify argument lists. As a result, the IDL compiler was not able to distinguish between arrays and functions by looking at the statement syntax. For example, the IDL statement

```
value = fish(5)
```

could either set the variable value equal to the sixth element of an array named fish, or set value equal to the result of passing the argument 5 to a function called fish.

To determine if it is compiling an array subscript or a function call, IDL checks its internal table of known functions. If it finds a function name that matches the unknown element in the command (fish, in the above example), it calls that function with the argument specified. If IDL does not find a function with the correct name in its table of known functions, it assumes that the unknown element is an array, and attempts to return the value of the designated element of that array. This rule generally gives the desired result, but it can be fooled into the wrong choice under certain circumstances, much to the surprise of the unwary programmer.

For this reason, versions of IDL beginning with version 5.0 use square brackets rather than parentheses for array subscripting. An array subscripted in this way is unambiguously interpreted as an array under all circumstances. In IDL 5.0 and later:

```
value = fish[5]
```

sets value to the sixth element of an array named fish.

Due to the large amount of existing IDL code written in the older syntax, as well as the ingrained habits of thousands of IDL users, IDL continues to allow the old syntax to be used, subject to the ambiguity mentioned above. That is, while

```
value = fish[5]
```

is unambiguous,

```
value = fish(5)
```

is still subject to the same ambiguity—and rules—that applied in IDL versions prior to version 5.0.

Since the older syntax has been used widely, you should not be surprised to see it from time to time. However, square brackets are the preferred form, and should be used for new code.

# Assignment Operations and Arrays

The following table shows the variations possible in expressions containing array and scalar subscripts. The result of the assignment operation depends upon the dimensionality of the subscript.

**Note** ─────────────────────────────────────────────────

A subscript structure can also be composed of a range of elements. If expression is scalar, it is inserted into the subarray. If *Variable*[*Range*] and Array are the same size, elements of Array specified by *Range* are inserted in *Variable*. It is illegal if *Variable*[*Range*] and Array are different sizes. See "Subscript Ranges" on page 319 for complete details. For information on when you should not use subscript ranges, see "Avoid Using Range Subscripts" on page 323.

─────────────────────────────────────────────────────────

| Syntax Structure | Description |
|---|---|
| *Variable*[Scalar*Subscripts*] = Scalar*Expression* | *Expression* is stored in a single element of *Variable*.<br><br>```arrOne = [1, 2, 3, 4, 5]```<br>```arrOne[2] = 9```<br>```PRINT, arrOne```<br>```        1    2    9    4    5``` |
| *Variable*[Scalar*Subscripts*] = Array*Expression* | *Expression* array is inserted in *Variable* array beginning at point indicated by subscript.<br><br>```arrOne = [1, 2, 3, 4, 5]```<br>```arrTwo = [11, 12]```<br>```arrOne[1] = ArrTwo```<br>```PRINT, arrOne```<br>```        1    11    12    4    5```<br><br>**Note -** An "out of range subscript" error will occur if you attempt to insert `arrTwo` elements into non-existent elements of `arrOne`. For example `arrOne[4] = ArrTwo` fails. |

*Table 15-2: Introduction to Subscript Expression Structures*

| Syntax Structure | Description |
|---|---|
| *Variable*[Array*Subscripts*] = Scalar*Expression* | *Expression* scalar is stored in designated elements of *Variable.* Other array elements are unchanged.<br><br>`arrOne = [1, 2, 3, 4, 5]`<br>`arrOne[[2, 4]] = 0`<br>`PRINT, arrOne`<br>`      1    2    0    4    0`<br>**Note -** Note the use of the double brackets. Attempting to assign zeros to the 3rd and 5th element of the array using<br>`arrOne[2, 4] = 0`<br>results in an error: "`Attempt to subscript ARRONE with <INT(4)> is out of range.`" IDL interprets this as attempting to modify a single element in the 3rd column and 5th row, which does not exist. |
| *Variable*[Array*Subscripts*] = *ArrayExpression* | Elements of *Expression* are stored in designated elements of *Variable.*<br><br>`arrOne = [1, 2, 3, 4, 5]`<br>`arrOne[[0, 2]] = [111,333]`<br>`PRINT, arrOne`<br>`     111    2    333    4    5`<br><br>**Note -** Elements of the subscript array that are negative, or greater than the highest subscript of the subscripted array, are clipped to the target array boundaries. For example,<br>`arrOne[[-1, 2]] = [111,333]`<br>has the same result as `arrOne[[0,2]]`. See "Clipping" on page 315 for details. |

*Table 15-2: Introduction to Subscript Expression Structures (Continued)*

**Note** ─────────────────────────────────────

Array operations are much more efficient than loops. See "Use Vector and Array Operations" on page 196 for details.

──────────────────────────────────────────────

# Using Scalar Values as Subscripts

Scalar quantities in IDL can be thought of as the first element of an array with one dimension. They can be subscripted with a zero reflecting the first and only position. Therefore,

```
; Assign the value of 5 to A.
A = 5

; Print the value of the first element of A.
PRINT, A[0]
```

IDL prints:

```
5
```

If we redefine the first element of A:

```
; Redefine the first element of A.
A[0] = 6

PRINT, A
```

IDL prints:

```
6
```

**Note** ————————————————————————————————————————————————

You cannot subscript a variable that has not yet been defined. Thus, if the variable B has not been previously defined, the statement:

```
B[0] = 9
```

will fail with the error "variable is undefined."

————————————————————————————————————————————————————————

## Subscripting Arrays Using Scalar Values

The subscripted variable can have either a scalar or array subscript with the form:

```
Variable[Subscripts] = Scalar_Expression
```

If the subscript expression is a scalar value, a single element of the specified array is set to the value of the scalar expression. The expression can be of any type and is converted, if necessary, to the type of the variable. The variable on the left side must be either an array or a file variable. Some examples of assigning scalar expressions to subscripted variables are:

```
; Set element 100 of data to value.
```

```
data[99] = 1.234999

; Store string in an array. aName must be a string array or an
; error will result.
aName[index] = 'Joe'

; Set element [X, Y] of the 2-dimensional array image to the value
; contained in pixel.
image[X, Y] = pixel
```

If the subscript expression is an array, the scalar value is stored in the elements of the array whose subscripts are elements of the subscript array. For example, the following statement zeroes the four specified elements of data: data[3], data[5], data[7] and data[9]:

```
data[[3, 5, 7, 9]] = 0
```

The subscript array is converted to integer type if necessary before use. Elements of the subscript array that are negative, or greater than the highest subscript of the subscripted array, are clipped to the target array boundaries. Note that a common error is to use a negative *scalar* subscript (e.g., A[-1]). Using this type of subscript causes an error. Negative *array* subscripts (e.g., A[[-1]]) do not cause errors.

When a subscripted variable reference appears in an expression, the values of the selected array elements are extracted. For example, the following statements extract the first two values from array by subscripting with a second array (indices) and store the values in the variable new_array:

```
array = [1.0, 2.0, 3.0, 4.0, 5.0]
indices = [0, 1]
new_array = array[indices]
PRINT, new_array
```

IDL prints:

```
1.0    2.0
```

See the following sections for more information on array subscripts and clipping.

# Using Arrays as Subscripts

Arrays can be used as subscripts to other arrays. Each element in the *subscript array* selects an element in the subscripted array. When subscript arrays are used in conjunction with subscript ranges (as discussed in "Combining Subscripts" on page 324), more than one element may be selected for each element of the subscript array.

If no subscript ranges are present, the length and dimensionality of the result is the same as that of the subscript expression. The type of the result is the same as that of the subscripted array. If only one subscript is present, all subscripts are interpreted as if the subscripted array has one dimension.

In the simple case of a single subscript array, the process can be described as follows:

$$
V[S] = \begin{cases} V_{S_i} & \text{if} & 0 \le S_i < n \\ V_0 & \text{if} & S_i < 0 \\ V_{n-1} & \text{if} & S_i \ge n \end{cases} \qquad \text{for } 0 \le i < m
$$

Here, the vector *V* has *n* elements, and the subscript array *S* has *m* elements. The result *V[S]* has the same dimensionality and number of elements as *S*. If the subscript expression applied to the variable is an array and an array appears on the right side of the statement:

```
Variable[Array] = Array
```

then elements from the right side are stored in the elements designated by the subscript vector. Only those elements of the subscripted variable whose subscripts appear in the subscript vector are changed. Note the use of array subscripts (double brackets). For example, the statement

```
B[[ 2, 4, 6 ]] = [4, 16, 36]
```

is equivalent to the following series of assignment statements:

```
B[2] = 4
B[4] = 16
B[6] = 36
```

For another example, consider the statements:

```
A = [6, 5, 1, 8, 4, 3]
B = [0, 2, 4, 1]
C = A[B]
PRINT, C
```

This produces the following output:

```
6       1       4       5
```

The first element of C is 6 because that is the number in the 0 position of A. The second is 1 because the value in B of 2 indicates the third position in A, and so on.

Subscript elements are interpreted as if the subscripted variable is a vector. For example, if A is a $10 \times n$ matrix, the element *A[i, j]* has the subscript *i+10\*j*.

When there is an array expression on the right, it is inserted into the array appearing on the left side of the equal sign starting at the point designated by the scalar subscript. For example, the following creates intArr, a 5 column by 2 row integer array of zeros. Insert array B into intArr beginning at the position designated by the scalar subscript (note the use of single brackets).

```
A = INTARR(5,2)
B = [222, 333, 444]
A[1] = B
PRINT, A
0      222     333     444       0
0        0       0       0       0
```

**Note**
The subscript array is converted to longword type before use if necessary. Regardless of structure, this subscript array is interpreted as a vector.

## Clipping

If an element of the subscript array is less than or equal to zero, the first element of the subscripted array is selected. If an element of the subscript array is greater than or equal to the last subscript in the subscripted array, the last element is selected.

**Note**
Elements of the subscript array that are negative or larger than the highest subscript are clipped to the target array boundaries. Note that a common error is to use a negative *scalar* subscript (e.g., A[-1]). Using this type of subscript causes an error. Negative *array* subscripts (e.g., A[[-1]]) do not cause errors.

This *clipping* of out of bounds elements can be disabled within a routine by using the STRICTARRSUBS option to the COMPILE_OPT statement. (See the documentation for "COMPILE_OPT" (*IDL Reference Guide*) for details.) If STRICTARRSUBS is in force, then array subscripts that refer to out of bounds elements will instead cause IDL to issue an error and stop execution, just as an out-of-range scalar subscript does.

# Examples Using Arrays as Subscripts

One way to create a square $n \times n$ identity matrix is as follows:

```
A = FLTARR(N, N)
A[INDGEN(N) * (N + 1)] = 1.0
```

The expression `INDGEN(N)*(N + 1)` results in a vector containing the subscripts of the diagonal elements `[0, N+1, 2N+2, ..., (N-1)*(N+1)]`. The following statements create a 10x10 identity matrix:

```
A = FLTARR(10, 10)
A[INDGEN(10) * 11] = 1
```

Yet another way is to use two array subscripts. The statements:

```
A = FLTARR(N, N)
A[INDGEN(N), INDGEN(N)] = 1.0
```

create the array subscripts `[[0,0], [1,1], ..., [n-1, n-1]]`.

Assume the variable A is a 10 by 10 array. Here, the subscripts of the diagonal elements (`A[0,0]`, `A[1,1]`, ..., `A[9, 9]`) are equal to 0, 11, 22, …, 99. The elements of the vector `INDGEN(10)*11` also are equal to 0, 11, 22, ..., 99, so the expression `A[INDGEN(10) * 11]` yields a 10-element vector containing to the diagonal elements of A.

# Conditionally Altering Array Elements

The WHERE function can be used to select array elements that meet certain conditions. For example, the statement:

```
data[WHERE(data LT 0)] = -1
```

sets all negative elements of data to -1 without changing the positive elements. The result of the function, WHERE(data LT 0), is a vector composed of the subscripts of the negative elements of data. Using this vector as a subscript changes only the negative elements.

Similarly, the WHERE function can be used to select elements of an array using expressions similar to A[WHERE(A GT 0)], which results in a vector composed only of the elements of A that are greater than 0.

The following statements create and display a 5x5 identity matrix, which consists of ones along a diagonal, and zeros everywhere else:

```
A = FLTARR(5, 5)
A[INDGEN(5) * 6] = 1
PRINT, A
```

The following statement sets elements of A with values of zero or less to -1:

```
A[WHERE(A LE 0)] = -1
PRINT, A
```

In this example, assume that the vector data contains data elements and that a data drop-out is denoted by a negative value. In addition, assume that there are never two or more adjacent drop-outs. The following statements replace all drop-outs with the average of the two adjacent good points:

```
; Subscript vector of drop-outs.
bad = WHERE(data LT 0)

; Replace drop-outs with average of previous and next point.
data[bad] = (data[bad - 1] + data[bad + 1]) / 2
```

In this example, the following actions are performed:

- We use the LT (less than) operator to create an array, with the same dimensions as data, that contains a 1 for every element of data that is less than zero and a zero for every element of data that is zero or greater. We use this "drop-out array" as a parameter for the WHERE function, which generates a vector that contains the one-dimensional *subscripts* of the elements of the drop-out array that are nonzero. The resulting vector, stored in the variable bad, contains the subscripts of the elements of data that are less than zero.

- The expression data[bad - 1] is a vector that contains the subscripts of the points immediately preceding the drop-outs; while similarly, the expression data[bad + 1] is a vector containing the subscripts of the points immediately after the drop-outs.

- The average of these two vectors is stored in data[bad], the points that originally contained drop-outs.

**Note**

Also see "Example—Using Array Operators and WHERE" on page 197 for an additional example.

# Subscript Ranges

Subscript ranges are used to select a subarray from an array by giving the starting and ending subscripts of the subarray in each dimension. Subscript ranges can be combined with scalar and array subscripts and with other subscript ranges. Any rectangular portion of an array can be selected with subscript ranges.

**Note** —————

Processing subscript ranges is inefficient. When possible, use an array or scalar subscript instead of specifying a subscript range where the beginning and ending subscripts are separated by the colon character. See "Avoid Using Range Subscripts" on page 323 for details.

There are six types of subscript ranges:

| Subscript Format | Description |
|---|---|
| [*] | All elements of a dimension. |
|  | This form is used with multidimensional arrays to select all elements along the dimension. For example, if `arr` is a 10-column by 12-row array, `arr[*, 11]` is the last row of `arr`, composed of elements `[arr[0,11], arr[1,11], ..., arr[9,11]]`, and is a 10-element row vector. Similarly, `arr[0, *]` is the first column of `arr`, `[arr[0,0], arr[0,1],..., arr[0,11]]`, and its dimensions are 1 column by 12 rows. |
| [$e_0$:$e_1$] | Subscript range from $e_0$ to $e_1$. |
|  | This denotes all elements whose subscripts range from the expression $e_0$ through $e_1$ ($e_0$ must not be greater than $e_1$). For example, if the variable `vec` is a 50-element vector, `vec[5:9]` is a five-element vector composed of `vec[5]` through `vec[9]`. |

*Table 15-3: Subscript Range Forms*

| Subscript Format | Description |
|---|---|
| [$e_0$:*] | A range from given element to the last element of dimension. |
| | This denotes all elements from a given element to the last element of the dimension. If the variable vec is a 50-element vector, vec[10:*] is a 40-element vector made from vec[10] through vec[49]. |
| [ $e_0$:$e_1$:$e_2$ ] | Every $e_2$th element in a range of subscripts from $e_0$ to $e_1$. |
| | This denotes every $e_2$th element within the range of subscripts $e_0$ through $e_1$ ( $e_0$ must not be greater than $e_1$). $e_2$ is referred to as the subscript *stride*. The stride value must be greater than or equal to 1. If it is set to the value 1, the resulting subscript expression is identical in meaning to [$e_0$:$e_1$], as described above. For example, if the variable vec is a 50-element vector, vec[5:13:2] is a five-element vector composed of vec[5], vec[7], vec[9], vec[11], and vec[13]. |
| [$e_0$:*:$e_2$] | Every $e_2$th element from element $e_0$ to the end of dimension. |
| | This denotes every $e_2$th element from a given element to the last element of the dimension, written as [$e_0$:*:$e_2$] where $e_2$ is referred to as the subscript stride. The stride value must be greater than or equal to 1. If it is set to the value 1, the resulting subscript expression is identical in meaning to [$e_0$:*], as described above. If the variable vec is a 50-element vector, vec[10:*:4] is a 10-element vector made from every fourth element between vec[10] through vec[49]. |
| [$n$] | A simple subscript. |
| | When used with multidimensional arrays, simple subscripts specify only elements with subscripts equal to the given subscript in that dimension. |

*Table 15-3: Subscript Range Forms (Continued)*

Multidimensional subarrays can be specified using any combination of the above forms. For example, if arr is a 10x10 array, arr[*, 0:4] is made from all columns of rows 0 to 4 of arr or a 10-column, 5-row array.

# Dimensionality of Subarrays

The dimensions of an extracted subarray are determined by the size in each dimension of the subscript range expression. In general, the number of dimensions is equal to the number of subscripts and subscript ranges. The size of the *n*-th dimension is equal to one if a simple subscript was used to specify that dimension in the subscript; otherwise, it is equal to the number of elements selected by the corresponding range expression.

Degenerate dimensions (trailing dimensions with a size of one) are removed. If arr is a 10-column by 12-row array, the expression arr[*,11] results in a row vector with a single dimension. (The result of the expression is a 10-column by 1-row array; the last dimension is degenerate and is removed.) On the other hand, the expression arr[0, *] became a column vector with dimensions of [1, 12], showing that the structure of columns is preserved because the dimension with a size of one does not appear at the end.

To see this, enter the following statements in IDL:

```
arr = INDGEN(10,12)
HELP, arr
HELP, arr[*,11]
HELP, arr[0,*]
```

In the following examples, vec is a 50-element floating-point vector, and arr is a 10-column by 12-row integer array. Some typical subscript range expressions are as follows:

```
vec = FINDGEN(50)
arr = INDGEN(10,12)

; Elements 5 through 10 of vec, a six-element vector.
vec[5:10]

; A three-element vector.
vec[I - 1:I + 1]

; The same vector.
[vec[I - 1], vec[I], vec[I + 1]]

; Elements from vec[4] to the end, a 46-element (50-4) vector.
vec[4:*]

; Values of the elements with even subscripts in vec.
vec[0:*:2]
```

```
; Values of the elements with odd subscripts in vec:
vec[1:*:2]

; The fourth column of arr, a 1 column by 12 row vector.
arr[3, *]

; The first row of arr, a 10-element row vector. Note, the last
; dimension was removed because it was degenerate.
[arr[3, 0], arr[3, 1], ..., arr[3, 11]]
arr[*, 0]

; The nine-point neighborhood surrounding arr[X,Y], a 3 by 3 array.
arr[X - 1:X + 1, Y - 1:Y + 1]

; Three columns of arr, a 3 by 12 subarray:
arr[3:5,*]
```

To insert the contents of an array called A into array B, starting at point B[13, 24], use the following statement:

```
B[13, 24] = A
```

If A is a 5-column by 6-row array, elements B[13:17, 24:29] are replaced by the contents of array A.

In the next example, a subarray is moved from one position to another:

```
B[100, 200] = B[200:300, 300:400]
```

A subarray of B, specifically the columns 200 to 300 and rows 300 to 400, is moved to columns 100 to 200 and rows 200 to 300, respectively.

Assuming the variable B is a $512 \times 512$-byte array, some examples are as follows:

```
; Store 1 in every element of the i-th row.
array[*, I] = 1

; Store 1 in every element of the j-th column.
array[J, *] = 1

; Zero all the rows of columns 200 through 220 of array.
array[200:220, *] = 0

; Store the value 100 in all the elements of array.
array[*] = 100
```

# Avoid Using Range Subscripts

It is possible to use range subscripts in an assignment statement, however, when possible, you should avoid using range subscripts in favor of using scalar or array subscripts. This type of assignment statement takes the following form:

```
Variable[Subscript_Range] = Expression
```

A subscript range specifies a beginning and ending subscripts, which are separated by the colon character. An ending subscript equal to the size of the dimension minus one can be written as *. For example, arr[I:J] denotes those points in the vector arr with subscripts between I and J inclusive. I must be less than or equal to J and greater than or equal to zero. J denotes the points in arr from arr[I] to the last point and must be less than the size of the dimension arr [I:*]. See "Subscript Ranges" on page 319 for more details on subscript ranges.

When possible, you should avoid using range subscripts in favor of using scalar or array subscripts. In the following example, the array elements of X are inserted into array A. The slow way uses subscript ranges, specifying the insertion of X array elements into the 5th through 7th elements of A. The fast way uses a scalar subscript specifying the first element (the 5th) to be replaced with the elements of A.

```
A = INTARR(10)
X = [1,1,1]
PRINT, 'A = ', A
; Slow way:
t = SYSTIME(1) & FOR i=0L,100000 DO A[4:6] = X &
   PRINT,'Slow way: ', SYSTIME(1)-t
PRINT, 'A = ', A
; Correct way is 4 times faster!!:
t = SYSTIME(1) & FOR i=0L,100000 DO A[4] = X &
   PRINT, 'Fast way: ', SYSTIME(1)-t
PRINT, 'A = ', A
```

IDL prints:

```
A =   0    0    0    0    0    0    0    0    0    0
Slow way:       0.47000003
A =   0    0    0    0    1    1    1    0    0    0
Fast way:       0.12100005
A =   0    0    0    0    1    1    1    0    0    0
```

The statement A[4] = X, where X is a three-element array, causes IDL to start at index 4 of array A, and replace the next three elements in A with the elements in X. Because of the way it is implemented in IDL, A[4] = X is much more efficient than A[4:6] = X.

# Combining Subscripts

Subscript arrays can be combined with subscript ranges, simple scalar subscripts, and other subscript arrays.

When IDL encounters a multidimensional subscript expression that contains one or more subscript arrays, ranges, or scalars, it builds a subscript array by processing each element in the subscript expression from left to right. The resulting subscript array is then applied to the variable to be subscripted. As with other subscript operations, trailing degenerate dimensions (those with a size of 1) are eliminated.

## Subscript Ranges

When combining a subscript array with a subscript range, the result is an array of subscripts constructed by combining each element of the subscript array with each member of the subscript range. Combining an *n*-element array with an *m*-element subscript range yields an *nm*-element subscript. Each dimension of the result is equal to the number of elements in the corresponding subscript array or range.

For example, the expression A[[1, 3, 5], 7:9] is a nine-element, $3 \times 3$ array composed of the following elements:

$$\begin{bmatrix} A_{1,7} & A_{3,7} & A_{5,7} \\ A_{1,8} & A_{3,8} & A_{5,8} \\ A_{1,9} & A_{3,9} & A_{5,9} \end{bmatrix}$$

Each element of the three-element subscript array [1, 3, 5] is combined with each element of the three-element range (7, 8, 9).

Another example shows the common process of zeroing the edge elements of a two-dimensional $n \times m$ array:

```
; Zero the first and last rows.
A[*, [0, M-1]] = 0

; Zero the first and last columns.
A[[0, N - 1], *] = 0
```

## Other Subscript Arrays

When combining two subscript arrays, each element of the first subscript array is combined with the corresponding element of the second subscript array. The two subscript arrays must have the same number of elements. The resulting subscript array has the same number of elements as its constituents. For example, the expression A[[1, 3], [5, 9]] yields the elements A[1,5] and A[3,9].

## Scalars

Combining an *n*-element subscript range or *n*-element subscript array with a scalar yields an *n*-element result. The value of the scalar is combined with each element of the range or array. For example, the expression A[[1, 3, 5], 8] yields the three-element vector composed of the elements A[1,8], A[3,8], and A[5,8]. The second dimension of the result is 1 and is eliminated because it is degenerate. The expression A[8, [1, 3, 5]] is the $1 \times 3$-column vector A[8,1], A[8,3], and A[8,5], illustrating that leading dimensions are not eliminated.

# Manipulating Arrays

IDL provides a variety of mechanisms for working with multidimensional data sets. Understanding these mechanisms requires a familiarity with linear algebra and the concept of a two-dimensional data set.

**Note** ─────────────────────────────────────────────

There are two terms commonly used to refer to two-dimensional data sets: *array* and *matrix*. People who work with images tend to call two-dimensional data sets arrays, while mathematicians tend to call two-dimensional data sets matrices. The terms are interchangeable, but the different conventions assumed by people who use them may lead to confusion.

─────────────────────────────────────────────────────

Consider a two-dimensional data set, with dimensions *m* and *n*. In a computer, the data from this data set is stored in a unidimensional set of memory addresses; what makes the data "two-dimensional" is the way the individual elements are indexed by the software that accesses the data in memory. This topic is discussed in detail in "Columns, Rows, and Array Majority" on page 332; if you are unsure of your understanding of the process of mapping multidimensional data into unidimensional computer memory, please read that section carefully.

There are two possible ways to depict a two-dimensional data set on paper — row by row or column by column. For example, the standard mathematical representation of an *m x n* data set is shown in Figure 15-1, with *m* rows and *n* columns:

$$\begin{bmatrix} A_{0,0} & A_{0,1} & \cdots & A_{0,n-1} \\ A_{1,0} & A_{1,1} & \cdots & A_{1,n-1} \\ \cdots & \cdots & \cdots & \cdots \\ A_{m-1,0} & A_{m-1,1} & \cdots & A_{m-1,n-1} \end{bmatrix}$$

*Figure 15-1: An m x n Array Represented in Mathematical Notation*

Here, the first dimension (*m*) represents the row index, and the second dimension (*n*) represents the column index. Thus, if the data set is represented using this notation, the term `Array[3,2]` refers to an element that is four rows down from the top row and three columns to the right of the leftmost row. (Note that indices are zero-based.)

Figure 15-4 depicts the standard image-processing representation of the same data set, with *m* columns and *n* rows:

$$
\begin{bmatrix}
A_{0,0} & A_{1,0} & \cdots & A_{m-1,0} \\
A_{0,1} & A_{1,1} & \cdots & A_{m-1,1} \\
\cdots & \cdots & \cdots & \cdots \\
A_{0,n-1} & A_{1,n-1} & \cdots & A_{m-1,n-1}
\end{bmatrix}
$$

*Figure 15-2: An m x n Array Represented in Image-processing Notation*

Here, the first dimension (*m*) represents the column index, and the second dimension (*n*) represents the row index. Thus, if the data set is represented using this notation, the term `Array[3,2]` refers to an element that is four columns to the right of the leftmost column and three rows down from the top row. This is the representation used by IDL.

It is important to understand that these are two views of the *same* data; all that has changed is the notational convention applied. Why is this notational convention important? Because when reading or writing data in a two-dimensional data set, performance improves if elements that are contiguous in the computer's memory are accessed consecutively. Incrementing the index of the first dimension by one shifts one "slot" in computer memory, whereas incrementing the index of the second dimension by one shifts a number of "slots" at least as large as the size of the first dimension.

**Note**

The terms *column-major* and *row-major* are commonly used to define which dimension of a two-dimensional array represents the column index and which represents the row index. These terms are defined and discussed in detail in "Columns, Rows, and Array Majority" on page 332.

# Transposing Arrays

You should be aware that many numerical algorithms — especially those that are written in a row-major language such as C or C++ — assume data is indexed (row, column). Since IDL assumes data is indexed (column, row), it is important to keep

this distinction in mind. In order to work with data indexed (row, column), you can use IDL's TRANSPOSE function to interchange the order of the indices.

Note that it is possible for an array to be indistinguishable from its transpose. In this case the number of columns and rows are identical and there is a symmetry between the rows of the array and the columns of its transpose. Arrays satisfying this condition are said to be *symmetric*. When dealing with symmetric arrays the use of the TRANSPOSE function is unnecessary, since $A^T = A$.

# Multiplying Arrays

IDL has two operators used to multiply arrays. To illustrate the difference between the two operators, consider the following two arrays:

```
; A 3-column by 2-row array:
A = [ [0, 1, 2],$
      [3, 4, 5] ]

; A 2-column by 3-row array:
B = [ [0, 1],$
      [2, 3],$
      [4, 5] ]
```

## The # Operator

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

For example, consider the arrays defined above:

$$A = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix}, B = \begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix}$$

We obtain the elements of A # B as follows:

$$\begin{bmatrix} A_{0,0}B_{0,0} + A_{0,1}B_{1,0} & A_{1,0}B_{0,0} + A_{1,1}B_{1,0} & A_{2,0}B_{0,0} + A_{2,1}B_{1,0} \\ A_{0,0}B_{0,1} + A_{0,1}B_{1,1} & A_{1,0}B_{0,1} + A_{1,1}B_{1,1} & A_{2,0}B_{0,1} + A_{2,1}B_{1,1} \\ A_{0,0}B_{0,2} + A_{0,1}B_{1,2} & A_{1,0}B_{0,2} + A_{1,1}B_{1,2} & A_{2,0}B_{0,2} + A_{2,1}B_{1,2} \end{bmatrix}$$

Or, using the actual values from the arrays:

$$\begin{bmatrix} (0)(0) + (3)(1) & (1)(0) + (4)(1) & (2)(0) + (5)(1) \\ (0)(2) + (3)(3) & (1)(2) + (4)(3) & (2)(2) + (5)(3) \\ (0)(4) + (3)(5) & (1)(4) + (4)(5) & (2)(4) + (5)(5) \end{bmatrix}$$

Therefore, when we issue the following command:

```
PRINT, A#B
```

IDL prints:

```
   3           4           5
   9          14          19
  15          24          33
```

**Tip** —————————————————————————————————————————————

If one or both of the arrays are also transposed, such as TRANSPOSE(A) # B, it is more efficient to use the MATRIX_MULTIPLY function, which does the transpose simultaneously with the multiplication.

————————————————————————————————————————————————————

### Note on the Definition of Matrix Multiplication

While the definition of the IDL # operator may *appear* to be at odds with the standard mathematical definition of matrix multiplication — namely, that the operator multiplies each row of the first matrix by each column of the second matrix — this is a case of slightly imprecise terminology. The confusion arises from the mappings of the words "row" and "column" — which refer to elements in a two-dimensional entity called an *array* or a *matrix* — to the one-dimensional vector of values stored in computer memory. In reality, what the matrix multiplication operator does is multiply the elements of the first *dimension* of the first array/matrix by the elements of the second *dimension* of the second array/matrix. IDL's convention is to consider the first dimension to be the column and the second dimension to be the row, whereas the standard mathematical convention considers the first dimension to be the row and the second dimension to be the column. For a more complete discussion of this topic, see "Columns, Rows, and Array Majority" on page 332.

## The ## Operator

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

For example, consider the arrays defined above:

$$A = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix}, B = \begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix}$$

We obtain the elements of A ## B as follows:

$$\begin{bmatrix} A_{0,0}B_{0,0} + A_{1,0}B_{0,1} + A_{2,0}B_{0,2} & A_{0,0}B_{1,0} + A_{1,0}B_{1,1} + A_{2,0}B_{1,2} \\ A_{0,1}B_{0,0} + A_{1,1}B_{0.1} + A_{2,1}B_{0,2} & A_{0,1}B_{1,0} + A_{1,1}B_{1,1} + A_{2,1}B_{1,2} \end{bmatrix}$$

Or, using the actual values from the arrays:

$$\begin{bmatrix} (0)(0) + (1)(2) + (2)(4) & (0)(1) + (1)(3) + (2)(5) \\ (3)(0) + (4)(2) + (5)(4) & (3)(1) + (4)(3) + (5)(5) \end{bmatrix}$$

Therefore, when we issue the following command:

```
PRINT, A##B
```

IDL prints:

```
10          13
28          40
```

## Multiplying Vectors

When using the # and ## operators to multiply vectors, note the following:

- For A # B, where A and B are vectors, IDL performs A # TRANSPOSE(B). In this case, C = A # B is a matrix with Cij = Ai Bj. Mathematically, this is equivalent to the *outer product*, usually denoted by $A \otimes B$.

- For A ## B, where A and B are vectors, IDL performs TRANSPOSE(A) ## B. In this case, C = A ## B is a matrix with Cij = Bi Aj.

- To compute the *dot product*, usually denoted by $A \cdot B$, use TRANSPOSE(A) # B.

## Notes on the # and ## Operators

Note the following with regard to the array multiplication operators:

- The # and ## operators are order specific.

- A # B = B ## A

- A # B = $(B^T \# A^T)^T$

## Routines for Multiplying Arrays

The MATRIX_MULTIPLY and MATRIX_POWER routines are also available:

- MATRIX_MULTIPLY calculates the value of the # operator applied to two (possibly transposed) arrays. See "MATRIX_MULTIPLY" (*IDL Reference Guide*) for details.

- MATRIX_POWER computes the product of a matrix with itself. See "MATRIX_POWER" (*IDL Reference Guide*) for details.

**Note** ───────────────────────────────────────────────

Also see "Array Manipulation" (*IDL Quick Reference*) for a list of other array manipulation routines.

─────────────────────────────────────────────────────────

# Columns, Rows, and Array Majority

Computer hardware does not directly support the concept of multidimensional arrays. Computer memory is unidimensional, providing memory addresses that start at zero and increase serially to the highest available location. Multidimensional arrays are therefore a software concept: software (IDL in this case) maps the elements of a multi-dimensional array into a contiguous linear span of memory addresses. There are two ways that such an array can be represented in one-dimensional linear memory. These two options, which are explained below, are commonly called *row major* and *column major.* All programming languages that support multidimensional arrays must choose one of these two possibilities. This choice is a fundamental property of the language, and it affects how programs written in different languages share data with each other.

Before describing the meaning of these terms and IDL's relationship to them, it is necessary to understand the conventions used when referring to the dimensions of an array. For mnemonic reasons, people find it useful to associate higher level meanings with the dimensions of multi-dimensional data. For example, a 2-D variable containing measurements of ozone concentration on a uniform grid covering the earth might associate latitude with the first dimension, and longitude with the second dimension. Such associations help people understand and reason about their data, but they are not fundamental properties of the language itself. It is important to realize that no matter what meaning you attach to the dimensions of an array, IDL is only aware of the number of dimensions and their size, and does not work directly in terms of these higher order concepts. Another way of saying this is that `arr[d1, d2]` addresses the same element of variable arr no matter what meaning you associate with the two dimensions.

In the IDL world, there are two such conventions that are widely used:

- In image processing, the first dimension of an image array is the column, and the second dimension is the row. IDL is widely used for image processing, and has deep roots in this area. Hence, the dominant convention in IDL documentation is to refer to the first dimension of an array as the column and the second dimension as the row.

- In the standard mathematical notation used for linear algebra, the first dimension of an array (or *matrix*) is the row, and the second dimension is the column. Note that this is the exact opposite of the image processing convention.

In computer science, the way array elements are mapped to memory is always defined using the mathematical [*row*, *column*] notation. Much of the following discussion utilizes the *m x n* array shown in Figure 15-3, with *m* rows and *n* columns:

$$\begin{bmatrix} A_{0,\,0} & A_{0,\,1} & \cdots & A_{0,\,n-1} \\ A_{1,\,0} & A_{1,\,1} & \cdots & A_{1,\,n-1} \\ \cdots & \cdots & \cdots & \cdots \\ A_{m-1,\,0} & A_{m-1,\,1} & \cdots & A_{m-1,\,n-1} \end{bmatrix}$$

*Figure 15-3: An m x n array represented in mathematical notation.*

Given such a 2-dimensional matrix, there are two ways that such an array can be represented in 1-dimensional linear memory — either row by row (*row major)*, or column by column (*column major*):

- **Contiguous First Dimension (Column Major)**: In this approach, all elements of the first dimension (*m* in this case) are stored contiguously in memory. The 1-D linear address of element $A_{d1,\,d2}$ is therefore given by the formula `(d2*m + d1)`. As you move linearly through the memory of such an array, the first (leftmost) dimension changes the fastest, with the second dimension (*n*, in this case) incrementing every time you come to the end of the first dimension:

  $A_{0,0}$, $A_{1,0}$, …, $A_{m-1,0}$, $A_{0,1}$, $A_{1,1}$, …, $A_{m-1,1}$, …

  Computer languages that map multidimensional arrays in this manner are called *column major,* following the mathematical [*row*, *column*] notation. IDL and Fortran are both examples of column-major languages.

- **Contiguous Second Dimension (Row Major)**: In this approach, all elements of the second dimension (*n*, in this case) are stored contiguously in memory. The 1-D linear address of element $A_{d1,\,d2}$ is therefore given by the formula `(d1*n + d2)`. As you move linearly through the memory of such an array, the second dimension changes the fastest, with the first dimension (*m* in this case) incrementing every time you come to the end of the second dimension:

  $A_{0,0}$, $A_{0,1}$, …, $A_{0,n-1}$, $A_{1,0}$, $A_{1,1}$, …, $A_{1,n-1}$, …

  Computer languages that map multidimensional arrays in this manner are known as *row major*. Examples of row-major languages include C and C++.

The terms *row major* and *column major* are widely used to categorize programming languages. It is important to understand that when programming languages are discussed in this way, the mathematical convention — in which the first dimension represents the row and the second dimension represents the column — is used. If you use the image-processing convention — in which the first dimension represents the column and the second dimension represents the row — you should be careful to make note of the distinction.

**Note** ─────────────────────────────────────────────────────────

IDL users who are comfortable with the IDL image-processing-oriented array notation [*column*, *row*] frequently follow the reasoning outlined above and incorrectly conclude that IDL is a row-major language. The often-overlooked cause of this mistake is that the standard definition of the terms *row major* and *column major* assume the mathematical [*row*, *column*] notation. In such cases, it can be helpful to look beyond the row/column terminology and think in terms of which dimension is contiguous in memory.

─────────────────────────────────────────────────────────────────

Note that the *m x n* array discussed above could be represented with equal accuracy as having *m* columns and *n* rows, as shown in Figure 15-4. This corresponds to the image-processing [*column*, *row*] notation. It's important to note that while the representation shown is the transpose of the representation in Figure 15-3, the data stored in the computer memory are identical. Only the two-dimensional representation, which takes its form from the notational convention used, has changed.

$$
\begin{bmatrix}
A_{0,0} & A_{1,0} & \cdots & A_{m-1,0} \\
A_{0,1} & A_{1,1} & \cdots & A_{m-1,1} \\
\cdots & \cdots & \cdots & \cdots \\
A_{0,n-1} & A_{1,n-1} & \cdots & A_{m-1,n-1}
\end{bmatrix}
$$

*Figure 15-4: An m x n array represented in image-processing notation.*

IDL's choice of column-major array layout reflects its roots as an image processing language. The fact that the elements of the first dimension are contiguous means that the elements of each row of an image array (using [*column*, *row*] notation, as shown in Figure 15-4) are contiguous. This is the order expected by most graphics hardware, providing an efficiency advantage for languages that naturally store data that way.

Also, this ordering minimizes virtual memory overhead, since images are accessed linearly.

It should be clear that the higher-level meanings associated with array dimensions (row, column, latitude, longitude, *etc.*) are nothing more than a human notational device. In general, you can assign any meaning you wish to the dimensions of an array, and as long as your use of those dimensions is consistent, you will get the correct answer, regardless of the order in which IDL chooses to store the actual array elements in computer memory. Thus, it is usually possible to ignore these issues. There are times however, when understanding memory layout can be important:

**Sharing Data With Other Languages —** If binary data written by a row major language is to be input and used by IDL, transposition of the data is usually required first. Similarly, if IDL is writing binary data for use by a program written in a row major language, transposition of the data before writing (or on input by the other program) is often required.

**Calling Code Written In Other Languages —** When passing IDL data to code written in a row major language via dynamic linking (CALL_EXTERNAL, LINKIMAGE, DLMs), it is often necessary to transpose the data before passing it to the called code, and to transpose the results.

**Matrix Multiplication —** Understanding the difference between the IDL # and ## operators requires an understanding of array layout. For a discussion of how the ordering of such data relates to IDL mathematics routines, see "Manipulating Arrays" on page 326.

**1-D Subscripting Of Multidimensional Array —** IDL allows you to index multidimensional arrays using a single 1-D subscript. For example, given a two dimensional 5x7 array, ARRAY[2,3] and ARRAY[17] refer to the same array element. Knowing this requires an understanding of the actual array layout in memory ($d2*m + d1$, or $3*5+2$, which yields 17).

**Efficiency —** Accessing memory in the wrong order can impose a severe performance penalty if your data is larger than the physical memory in your computer. Accessing elements of an array along the contiguous dimension minimizes the amount of memory paging required by the virtual memory subsystem of your computer hardware, and will therefore be the most efficient. Accessing memory across the non-contiguous dimension can cause each such access to occur on a different page of system memory. This forces the virtual memory subsystem into a cycle in which it must continually force current pages of memory to disk in order to make room for new pages, each of which is only momentarily accessed. This inefficient use of virtual memory is commonly known as *thrashing*.

# Chapter 16
# Structures

The following topics are covered in this chapter:

# Overview of Structures

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

## Named Structures

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

## Anonymous Structures

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

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

# Creating and Defining Structures

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

{*Structure_Name, Tag_Name$_1$ : Tag_Definition$_1$, ..., Tag_Name$_n$ : Tag_Definition$_n$*}

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

{*Tag_Name1 : Tag_Definition$_1$ , ..., Tag_Name$_n$ : Tag_Definition$_n$*}

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

Tag names may not be IDL Reserved Words, and must be unique within a given structure, although the same tag name can be used in more than one structure. Structure names and tag names follow the rules of IDL identifiers: they must begin with a letter; following characters can be letters, digits, or the underscore or dollar sign characters; and case is ignored.

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

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

{*Structure_Name* }

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

**Note** ———————————————————————————————————

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

———————————————————————————————————————————————

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

{*Structure_Name*, *expression$_1$*, ..., *expression$_n$*}

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

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

# Structure Inheritance

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

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

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

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

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

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

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

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

Structures that are inherited must be defined before the inheriting structure can be defined. If a structure inherits tags from another structure that is not yet defined, IDL will search for a routine to define the inherited structure as outlined in "Automatic Structure Definition" on page 354. If the inherited structure cannot be defined, definition of the new structure fails.

While structure inheritance can be used with any structure, it is most useful when dealing with *object class structures*. When the INHERITS specifier is used in a class structure definition, it has the added effect of defining the inheriting object as a *subclass* of the inherited class. For a discussion of object-oriented IDL programming, see Chapter 13, "Creating Custom Objects in IDL" (*Object Programming*).

# Example of Creating a Structure

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

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

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

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

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

or by using the CREATE_STRUCT function:

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

# Structure References

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

*Variable_Name.Tag_Name*

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

```
var.tag1.tag2
```

Each tag name, except possibly the last, must refer to a field that contains a structure.

## Subscripted Structure References

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

*Variable_Name.Tag_Name[Subscripts]*

*Variable_Name[Subscripts].Tag_Name...*

*Variable_Name[Subscripts].Tag_Name[Subscripts]*

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

*Structure_reference*:= *Variable_Name*{*[Subscripts]*}.*Tags*

*Tags*:= {*Tags.*}*Tag*

*Tag*:= *Tag_Name*{*[Subscripts]*}

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

```
A.B
A.B[N, M]
A[12].B
```

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

The semantics of storing into a structure field using subscript ranges is slightly
different than that of simple arrays. This is because the structure of arrays in fields are
fixed. See "Storing Into Array Fields" on page 345 for details.

# Examples of Structure References

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

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

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

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

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

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

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

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

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

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

# Using HELP with Structures

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

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

prints the following information:

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

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

# Parameter Passing with Structures

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

```
PRINT, A.name
```

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

```
READ, A.name
```

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

```
;Copy type and attributes to variable.
B = A.name

;Read into a simple variable.
READ, B

;Store result into field.
A.name = B
```

## Storing Into Array Fields

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

**VAR.ARRAY_TAG** = *Scalar_Expression*

All elements of VAR.tag are set to *Scalar_Expression*. For example:

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

**VAR.TAG** = *Array_Expression*

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

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

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

**VAR.TAG**[*Subscript*] = *Scalar_Expression*

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

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

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

**VAR.TAG**[*Subscript*] = *Array_Expression*

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

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

**VAR.TAG**[*Subscript_Range*] = *Scalar_Expression*

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

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

**VAR.TAG**[*Subscript_Range*] = *Array_Expression*

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

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

# Arrays of Structures

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

## Creating an Array of Structures

The easiest way to create an array of structures is to use the REPLICATE function. The first parameter to REPLICATE is a reference to the structure of each element. Using the example in "Examples of Structure References" on page 343 and assuming the STAR structure has been defined, an array containing 100 elements of the structure is created with the following statement:

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

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

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

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

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

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

## Examples of Arrays of Structures

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

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

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

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

;Prints name field of all 100 elements of cat, separated by commas
```

```
;(the last field has a trailing comma).
PRINT, cat.name + ','

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

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

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

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

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

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

# Structure Input/Output

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

## Formatted Input/Output with Structures

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

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

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

## Unformatted Input/Output with Structures

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

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

For example, the structure:

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

occupies four bytes on a machine where short integers must begin on an even byte boundary. When using ASSOC, a padding byte is added after field t1 to cause the integer field t2 to begin on an even-byte boundary. For more information, see "ASSOC" (*IDL Reference Guide*).

# Strings

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

# String Length Issues

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

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

This statement will truncate names longer than 20 characters and will pad with blanks those names shorter than 20 characters. The structure or structure array then can be output in a format suitable to be read by C or FORTRAN programs.

For example, to read into the cat array from a file in which each name field occupies 26 bytes, use the following statements.

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

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

# Advanced Structure Usage

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

> *Variable_Name*.(*Tag_Index*)... ... ...

The *Tag_Index* ranges from zero to the number of fields minus one.

**Note**

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

## Number of Structure Tags

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

## Names of Structure Tags

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

## Example

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

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

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

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

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

**Note** ───────────────────────────────────────────────

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

─────────────────────────────────────────────────────────

# Automatic Structure Definition

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

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

**Note** —————————————————————————————————————————————————

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

—————————————————————————————————————————————————————————

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

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

IDL prints:

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

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

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

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

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

IDL prints:

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

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

# Relaxed Structure Assignment

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

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

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

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

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

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

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

1.  Any fields found in the destination structure that are not found in the source structure are "zeroed" (set to zero, the empty string, or a null pointer or object reference depending on the type of field).

2.  Any fields in the source structure that are not found in the destination structure are quietly ignored.

3.  Any fields that are found in both the source and destination structures are copied one at a time. If necessary, type conversion is done to make their types agree. If a field in the source structure has fewer data elements than the corresponding field in the destination structure, then the "extra" elements in the field in the destination structure are zeroed. If a field in the source structure has more elements than the corresponding field in the destination structure, the extra elements are quietly ignored.

Using STRUCT_ASSIGN, we can make the assignment that failed using the = operator:

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

IDL prints:

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

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

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

IDL prints:

```
** Structure DEST, 2 tags, length=6:
   A               INT       Array[2]
   C               INT              0
```

# Using Relaxed Structure Assignment

Why would you want to use Relaxed Structure Assignment? One case where this type of structure definition is very useful is in restoring object structures into an environment where the structure definition may have changed since the restored objects were saved.

Suppose you have created an application that saves data in structures. Your application may use the IDL SAVE routine to save the data structures to disk files. If you later change your application such that the definition of the data structures changes, you would not be able to restore your saved data into your application's framework without relaxed structure assignment. The RELAXED_STRUCTURE_ASSIGNMENT keyword to the RESTORE procedure allows you to make relaxed assignments in such cases.

To see how this works, try the following exercise:

1. Start IDL, create a named structure, and use the SAVE procedure to save it to a file:

   ```
   mystruct = { STR, A:10, B:20L, C:'a string' }
   SAVE, mystruct, FILE='test.dat'
   ```

2. Exit and restart IDL.

3. Create a new structure definition with the same name you used previously:

   ```
   newstruct = { STR, A:20L, B:10.0, C:'a string', D:ptr_new() }
   ```

4.  Attempt to restore the variable mystruct from the test.dat file:

    ```
    RESTORE, 'test.dat'
    ```

    IDL prints:

    ```
    % Wrong number of tags defined for structure: STR.
    % RESTORE: Structure not restored due to conflict with
              existing definition: STR.
    ```

5.  Now use relaxed structure definition when restoring:

    ```
    RESTORE, 'test.dat', /RELAXED_STRUCTURE_ASSIGNMENT
    ```

6.  Check the contents of mystruct:

    ```
    HELP, mystruct, /STRUCTURE
    ```

    IDL prints:

    ```
    ** Structure STR, 4 tags, length=20:
       A               LONG                10
       B               FLOAT           20.0000
       C               STRING    'a string'
       D               POINTER   <NullPointer>
    ```

The structure in the variable mystruct now uses the definition from the new version of the STR structure, but contains the data from the old (restored) structure. In cases where the data type of a field has changed, the data type of the old data element has been converted to the new data type. Fields in the new structure definition that do not correspond to fields in the old definition contain "zero" values (zeroes for numeric fields, empty strings for string fields, null pointer or references for pointer or reference fields).

# Chapter 17
# Pointers

The following topics are covered in this chapter:

# Overview of Pointers

In order to build linked lists, trees, and other dynamic data structures, it must be possible to access variables via lightweight references that may have more than one name. Further, these names might have different lifetimes, so the lifetime of the variable that actually holds the data must be separate from the lifetime of the tokens that are used to access it.

Beginning with IDL version 5, IDL includes a new *pointer* data type to facilitate the construction of dynamic data structures. Although there are similarities between IDL pointers and machine pointers as implemented in languages such as C, it is important to understand that they are not the same thing. IDL pointers are a high level IDL language concept and do not have a direct one-to-one mapping to physical hardware. Rather than pointing at locations in computer memory, IDL pointers point at *heap variables*, which are special dynamically allocated IDL variables. Heap variables are global in scope, and exist until explicitly destroyed.

## Running the Example Code

The example code used in this chapter is part of the IDL distribution. All of the files mentioned are located in the examples/doc/language subdirectory of the IDL distribution. By default, this directory is part of IDL's path; if you have not changed your path, you will be able to run the examples as described here. See "!PATH" (*IDL Reference Guide*) for information on IDL's path.

# Heap Variables

Heap variables are a special class of IDL variables that have global scope and explicit user control over their lifetime. They can be basic IDL variables, accessible via pointers, or objects, accessible via object references. (See Chapter 13, "Creating Custom Objects in IDL" (*Object Programming*) for more information on IDL objects.) In IDL documentation of pointers and objects, heap variables accessible via pointers are called *pointer heap variables*, and heap variables accessible via object references are called *object heap variables*.

**Note** ───────────────────────────────────────────────────────────────

Pointers and object references have many similarities, the strongest of which is that both point at heap variables. It is important to understand that they are not the same type, and cannot be used interchangeably. Pointers and object references are used to solve different sorts of problems. Pointers are useful for building dynamic data structures, and for passing large data around using a lightweight token (the pointer itself) instead of copying data. Objects are used to apply object oriented design techniques and organization to a system. It is, of course, often useful to use both in a given program.

───────────────────────────────────────────────────────────────────────

Heap variables are global in scope, but do not suffer from the limitations of COMMON blocks. That is, heap variables are available to all program units at all times. (Remember, however, that IDL variables containing pointers to heap variables are *not* global in scope and must be declared in a COMMON block if you want to share them between program units.)

Heap variables:

- Facilitate object oriented programming.

- Provide full support for Save and Restore. Saving a pointer or object reference automatically causes the associated heap variable to be saved as well. This means that if the heap variable contains a pointer or object reference, the heap variables they point to are also saved. Complicated self-referential data structures can be saved and restored easily.

- Are manipulated primarily via pointers or object references using built in language operators rather than special functions and procedures.

- Can be used to construct arbitrary, fully general data structures in conjunction with pointers.

**Note** ———————————————————————————————————————————

If you have used versions of IDL prior to version 5, you may be familiar with *handles*. Because IDL pointers provide a more complete and robust way of building dynamic data structures, we recommend that you use pointers rather than handles when developing new code. See Appendix I, "IDL API History" (*IDL Reference Guide*) for a discussion of our policy on language features that have been superseded in this manner.

# Creating Heap Variables

Heap variables can be created only by the pointer creation function PTR_NEW or the object creation function OBJ_NEW. (See Chapter 13, "Creating Custom Objects in IDL" (*Object Programming*) for a discussion of object creation.) Copying a pointer or object reference *does not* create a new heap variable. This is markedly different from the way IDL handles "regular" variables. For example, with the statement:

```
A = 1.0
```

you create a new IDL floating-point variable with a value of 1.0. The following statement:

```
B = A
```

creates a second variable with the same type and value as A.

In contrast, if you create a new heap variable with the following command:

```
C = PTR_NEW(2.0d)
```

the variable C contains not the double-precision floating-point value 2.0, but a pointer to a heap variable that contains that value. Copying the variable C with the following statement:

```
D = C
```

does not create another heap variable, but rather creates a second pointer to the same heap variable. In this example, the HELP command would reveal:

```
% At  $MAIN$
A                FLOAT    =        1.00000
B                FLOAT    =        1.00000
C                POINTER  = <PtrHeapVar1>
D                POINTER  = <PtrHeapVar1>
```

The variables C and D are both pointers to the same heap variable. (The actual name assigned to a heap variable is arbitrary.) Changing the value stored in the heap variable would be reflected when dereferencing either C or D (dereferencing is discussed in "Dereference" on page 369).

Destroying or redefining either C, D, or both variables would leave the contents of the heap variable unchanged. When all pointers or references to a given heap variable are destroyed, the heap variable still exists and holds whatever memory has been allocated for it. See "Heap Variable Leakage" on page 374 for further discussion. If the heap variable itself is destroyed, pointers to the heap variable may still exist, but will be invalid. See "Dangling References" on page 373.

# Saving and Restoring Heap Variables

The SAVE and RESTORE procedures work for heap variables just as they work for all other supported types. By default, when IDL saves a pointer or object reference in a save file, it recursively saves the heap variables that are referenced by that pointer or object reference. SAVE handles circular data structures correctly. You can build a large, complicated, self-referential data structure, and then save the entire construct with a call to SAVE to save the single pointer or object reference that points to the head of the structure. For example, you can save a pointer to the root of a binary tree and the entire tree will be saved.

The internal identifier of a given heap variable is dynamically allocated at run time, and will differ between IDL sessions. As a result, the RESTORE operation maps all saved pointers and object references to their new values in the current session.

In some cases, you may want to save the pointer or object reference, but *not* the heap variable that are referenced by that pointer or object reference. You can specify that the heap variable associated with a pointer or object reference not be saved using the HEAP_NOSAVE procedure or the HEAP_SAVE function. See the documentation for HEAP_SAVE for additional details.

# Pointer Heap Variables

*Pointer heap variables* are IDL heap variables that are accessible only via *pointers*. While there are many similarities between object references and pointers, it is important to understand that they are not the same type, and cannot be used interchangeably. Pointer heap variables are created using the PTR_NEW and PTRARR functions. For more information on objects, see Chapter 13, "Creating Custom Objects in IDL" (*Object Programming*).

# IDL Pointers

As illustrated above, you must use a special IDL routine to create a pointer to a heap variable. Two routines are available: PTR_NEW and PTRARR. Before discussing these functions, however, it is useful to examine the concept of a null pointer.

## Null Pointers

The *Null Pointer* is a special pointer value that is guaranteed to never point at a valid heap variable. It is used by IDL to initialize pointer variables when no other initializing value is present. It is also a convenient value to use at the end nodes in data structures such as trees and linked lists.

It is important to understand the difference between a null pointer and a pointer to an undefined or invalid heap variable. The second case is a valid pointer to a heap variable that does not currently contain a usable value. To make the difference clear, consider the following IDL statements:

```
;The variable A contains a null pointer.
A = PTR_NEW()
;The variable B contains a pointer to a heap variable with an
;undefined value.
B = PTR_NEW(/ALLOCATE_HEAP)

HELP, A, B, *B
```

IDL prints:

```
A               POINTER  = <NullPointer>
B               POINTER  = <PtrHeapVar1>
<PtrHeapVar1>   UNDEFINED = <Undefined>
```

The primary difference is that it is possible to write a useful value into a pointer to an undefined variable, but this is never possible with a null pointer. For example, attempt to assign the value 34 to the null pointer:

```
*A = 34
```

IDL prints:

```
% Unable to dereference NULL pointer: A.
% Execution halted at:  $MAIN$
```

Assign the value 34 to a previously-undefined heap variable:

```
*B = 34
PRINT, *B
```

IDL prints:

```
34
```

Similarly, the null pointer is not the same thing as the result of PTR_NEW(0). PTR_NEW(0) returns a pointer to a heap variable that has been initialized with the integer value 0.

# The PTR_NEW Function

Use the PTR_NEW function to create a single pointer to a new heap variable. If you supply an argument, the newly-created heap variable is set to the value of the argument. For example, the command:

```
ptr1 = PTR_NEW(FINDGEN(10))
```

creates a new heap variable that contains the ten-element floating point array created by FINDGEN, and places a pointer to this heap variable in ptr1.

Note that the argument to PTR_NEW can be of any IDL data type, and can include any IDL expression, including calls to PTR_NEW itself. For example, the command:

```
ptr2 = PTR_NEW({name:'', next:PTR_NEW()})
```

creates a pointer to a heap variable that contains an anonymous structure with two fields: the first field is a string, the second is a pointer. We will develop this idea further in the examples at the end of this chapter.

If you do not supply an argument, the newly-created pointer will be a null pointer. If you wish to create a new heap variable but do not wish to initialize it, use the ALLOCATE_HEAP keyword.

See "PTR_NEW" (*IDL Reference Guide*) for further details.

# The PTRARR Function

Use the PTRARR function to create an array of pointers of up to eight dimensions. By default, every element of the array created by PTRARR is set to the null pointer. For example:

```
;Create a 2 by 2 array of null pointers.
ptarray = PTRARR(2,2)

;Display the contents of the ptarray variable, and of the first
;array element.
HELP, ptarray, ptarray(0,0)
```

IDL prints:

```
PTARR            POINTER   = Array(2, 2)
<Expression>     POINTER   = <NullPointer>
```

If you want each element of the array to point to a new heap variable (as opposed to being a null pointer), use the ALLOCATE_HEAP keyword. Note that in either case, you will need to initialize the array with another IDL statement.

See "PTRARR" (*IDL Reference Guide*) for further details.

# Operations on Pointers

Pointer variables are not directly usable by many of the operators, functions, or procedures provided by IDL. You cannot, for example, do arithmetic on them or plot them. You can, of course, do these things with the heap variables referenced by such pointers, assuming that they contain appropriate data for the task at hand. Pointers exist to allow the construction of dynamic data structures that have lifetimes that are independent of the program scope they are created in.

There are 4 IDL operators that work with pointer variables: assignment, dereference, EQ, and NE. The remaining operators (addition, subtraction, etc.) do not make any sense for pointer types and are not defined.

Many non-computational functions and procedures in IDL do work with pointer variables. Examples are SIZE, N_ELEMENTS, HELP, and PRINT. It is worth noting that the only I/O allowed directly on pointer variables is default formatted output, where they are printed as a symbolic description of the heap variable they point at. This is merely a debugging aid for the IDL programmer—input/output of pointers does not make sense in general and is not allowed. Please note that this does *not* imply that I/O on the contents of non-pointer data held in heap variables is not allowed. Passing the contents of a heap variable that contains non-pointer data to the PRINT command is a simple example of this type of I/O.

## Assignment

Assignment works in the expected manner—assigning a pointer to a variable gives you another variable with the same pointer. Hence, after executing the statements:

```
A = PTR_NEW(FINDGEN(10))
B = A
HELP, A, B
```

A and B both point at the same heap variable and we see the output:

```
A               POINTER  = <PtrHeapVar1>
B               POINTER  = <PtrHeapVar1>
```

## Dereference

In order to get at the contents of a heap variable referenced by a pointer variable, you must use the *dereference operator*, which is * (the asterisk). The dereference operator precedes the variable dereferenced. For example, if you have entered the above assignments of the variables A and B:

```
PRINT, *B
```

IDL prints:

```
0.00000    1.00000    2.00000    3.00000    4.00000    5.00000
6.00000    7.00000    8.00000    9.00000
```

That is, IDL prints the contents of the heap variable pointed at by the pointer variable B.

## Dereferencing Pointer Arrays

Note that the dereference operator requires a *scalar* pointer operand. This means that if you are dealing with a pointer array, you must specify which element to dereference. For example, create a three-element pointer array, allocating a new heap variable for each element:

```
ptarr = PTRARR(3, /ALLOCATE_HEAP)
```

To initialize this array such that the heap variable pointed at by the first pointer contains the integer zero, the second the integer one, and the third the integer two, you would use the following statement:

```
FOR I = 0,2 DO *ptarr[I] = I
```

**Note** ───────────────────────────────────────────────────────

The dereference operator is dereferencing only element I of the array for each iteration. Similarly, if you wanted to print the values of the heap variables pointed at by the pointers in ptarr, you might be tempted to try the following:

```
PRINT, *ptarr
```

IDL prints:

```
% Expression must be a scalar in this context: PTARR.
% Execution halted at:   $MAIN$
```

───────────────────────────────────────────────────────────────

To print the contents of the heap variables, use the statement:

```
FOR I = 0, N_ELEMENTS(ptarr)-1 DO PRINT, *ptarr[I]
```

## Dereferencing Pointers to Pointers

The dereference operator can be applied as many times as necessary to access data pointed at indirectly via multiple pointers. For example, the statement:

```
A = PTR_NEW(PTR_NEW(47))
```

assigns to A a pointer to a pointer to a heap variable containing the value 47.

To print this value, use the following statement:

```
PRINT, **A
```

## Dereferencing Pointers within Structures

If you have a structure field that contains a pointer, dereference the pointer by prepending the dereference operator to the front of the structure name. For example, if you define the following structure:

```
struct = {data:'10.0', pointer:ptr_new(20.0)}
```

you would use the following command to print the value of the heap variable pointed at by the pointer in the pointer field:

```
PRINT, *struct.pointer
```

Defining pointers to structures is another common practice. For example, if you define the following pointer:

```
ptstruct = PTR_NEW(struct)
```

you would use the following command to print the value of the heap variable pointed at by the pointer field of the struct structure, which is pointed at by ptstruct:

```
PRINT, *(*pstruct).pointer
```

Note that you must dereference both the pointer to the structure and the pointer within the structure.

## Dereferencing the Null Pointer

It is an error to dereference the NULL pointer, an invalid pointer, or a non-pointer. These cases all generate errors that stop IDL execution. For example:

```
PRINT, *45
```

IDL prints:

```
% Pointer type required in this context: <INT(      45)>.
% Execution halted at:  $MAIN$
```

For example:

```
A = PTR_NEW() & PRINT, *A
```

IDL prints:

```
% Unable to dereference NULL pointer: A.
% Execution halted at:  $MAIN$
```

For example:

```
A = PTR_NEW(23) & PTR_FREE, A & PRINT, *A
```

IDL prints:

% Invalid pointer: A.

% Execution halted at:  $MAIN$

# Equality and Inequality

The EQ and NE operators allow you to compare pointers to see if they point at the same heap variable. For example:

```
;Make A a pointer to a heap variable containing 23.
A = PTR_NEW(23)

;B points at the same heap variable as A.
B = A

;C contains the null pointer.
C = PTR_NEW()

PRINT, 'A EQ B: ', A EQ B & $
PRINT, 'A NE B: ', A NE B & $
PRINT, 'A EQ C: ', A EQ C & $
PRINT, 'C EQ NULL: ', C EQ PTR_NEW() & $
PRINT, 'C NE NULL:', C NE PTR_NEW()
```

IDL prints:

```
A EQ B:    1
A NE B:    0
A EQ C:    0
C EQ NULL: 1
C NE NULL: 0
```

# Dangling References

If a heap variable is destroyed, any remaining pointer variable or object reference that still refers to it is said to contain a *dangling reference*. Unlike lower level languages such as C, dereferencing a dangling reference will not crash or corrupt your IDL session. It will, however, fail with an error message. For example:

```
;Create a new heap variable.
A = PTR_NEW(23)

;Print A and the value of the heap variable A points to.
PRINT, A, *A
```

IDL prints:

```
<PtrHeapVar13>         23
```

For example:

```
;Destroy the heap variable.
PTR_FREE, A

;Try to print again.
PRINT, A, *A
```

IDL prints:

```
% Invalid pointer: A.
% Execution halted at:  $MAIN$
```

There are several possible approaches to avoiding such errors. The best option is to structure your code such that dangling references do not occur. You can, however, verify the validity of pointers or object references before using them (via the PTR_VALID or OBJ_VALID functions) or use the CATCH mechanism to recover from the effect of such a dereference.

# Heap Variable Leakage

Heap variables are not reference counted—that is, IDL does not keep track of how many references to a heap variable exist, or stop the last such reference from being destroyed—so it is possible to lose access to them and the memory they are using. For example:

```
;Create a new heap variable.
A = PTR_NEW(23)

;Set the pointer A equal to the integer zero. The pointer to the
;heap variable created with the first command is lost.
A = 0
```

Use the HEAP_VARIABLES keyword to the HELP procedure to view a list of heap variables currently in memory:

```
HELP, /HEAP_VARIABLES
```

IDL prints:

```
<PtrHeapVar14>  INT       =        23
```

In this case, the heap variable <PtrHeapVar14> exists and has a value of 23, but there is no way to reference the variable. There are two options: manually create a new pointer to the existing heap variable using the PTR_VALID function (see "PTR_VALID" (*IDL Reference Guide*)), or do manual "Garbage Collection" and use the HEAP_GC command to destroy all inaccessible heap variables.

**Warning** ————————————————————————————————

Object reference heap variables are subject to the same problems as pointer heap variables. See "OBJ_VALID" (*IDL Reference Guide*) for more information.

————————————————————————————————————————————

The HEAP_GC procedure causes IDL to hunt for all unreferenced heap variables and destroy them. It is important to understand that this is a potentially computationally expensive operation, and should not be relied on by programmers as a way to avoid writing careful code. Rather, the intent is to provide programmers with a debugging aid when attempting to track down heap variable leakage. In conjunction with the VERBOSE keyword, HEAP_GC makes it possible to determine when variables have leaked, and it provides some hint as to their origin.

**Warning** ————————————————————————————————

HEAP_GC uses a recursive algorithm to search for unreferenced heap variables. If HEAP_GC is used to manage certain data structures, such as large linked lists, a

potentially large number of operations may be pushed onto the system stack. If so many operations are pushed that the stack runs out of room, IDL will crash.

---

General reference counting, the usual solution to such leaking, is too slow to be provided automatically by IDL, and careful programming can easily avoid this pitfall. Furthermore, implementing a reference counted data structure on top of IDL pointers is easy to do in those cases where it is useful, and such reference counting could take advantage of its domain specific knowledge to do the job much faster than the general case.

Another approach would be to write allocation and freeing routines—layered on top of the PTR_NEW and PTR_FREE routines—that keep track of all outstanding pointer allocations. Such routines might make use of pointers themselves to keep track of the allocated pointers. Such a facility could offer the ability to allocate pointers in named groups, and might provide a routine that frees all heap variables in a given group. Such an operation would be very efficient, and is easier than reference counting.

# Pointer Validity

Use the PTR_VALID function to verify that one or more pointer variables point to valid and currently existing heap variables, or to create an array of pointers to existing heap variables. If supplied with a single pointer as its argument, PTR_VALID returns TRUE (1) if the pointer argument points at a valid heap variable, or FALSE (0) otherwise. If supplied with an array of pointers, PTR_VALID returns an array of TRUE and FALSE values corresponding to the input array. If no argument is specified, PTR_VALID returns an array of pointers to all existing pointer heap variables. For example:

```
;Create a new pointer and heap variable.
A = PTR_NEW(10)

IF PTR_VALID(A) THEN PRINT, "A points to a valid heap variable." $
    ELSE PRINT, "A does not point to a valid heap variable."
```

IDL prints:

```
A points to a valid heap variable.
```

For example:

```
;Destroy the heap variable.
PTR_FREE, A

IF PTR_VALID(A) THEN PRINT, "A points to a valid heap variable." $
    ELSE PRINT, "A does not point to a valid heap variable."
```

IDL prints:

```
A does not point to a valid heap variable.
```

See "PTR_VALID" (*IDL Reference Guide*) for further details.

# Freeing Pointers

The PTR_FREE procedure destroys the heap variables pointed at by pointers supplied as its arguments. Any memory used by the heap variable is released, and the heap variable ceases to exist. PTR_FREE is the only way to destroy a pointer heap variable; if PTR_FREE is not called on a heap variable, it continues to exist until the IDL session ends, even if no pointers remain to reference it.

Note that the pointers themselves are not destroyed. Pointers that point to nonexistent heap variables are known as dangling references, and are discussed in more detail in "Dangling References" on page 373.

See "PTR_FREE" (*IDL Reference Guide*) for further details.

The HEAP_FREE procedure recursively frees all heap variables (pointers or objects) referenced by its input argument. This routine examines the input variable, including all array elements and structure fields. When a valid pointer or object reference is encountered, that heap variable is marked for removal, and then is recursively examined for additional heap variables to be freed. In this way, all heap variables that are referenced directly or indirectly by the input argument are located. Once all such heap variables are identified, HEAP_FREE releases them in a final pass. Pointers are released as if the PTR_FREE procedure was called. Objects are released as with a call to OBJ_DESTROY.

HEAP_FREE is recommended when:

- The data structures involved are highly complex, nested, or variable, and writing cleanup code is difficult and error prone.

- The data structures are opaque, and the code cleaning up does not have knowledge of the structure.

See "HEAP_FREE" (*IDL Reference Guide*) for further details.

# Pointer Examples

Pointers are useful in building dynamic memory structures, such as linked lists and trees. The following examples demonstrate how pointers are used to build several types of dynamic structures. Note that the purpose of these examples is to illustrate simply and clearly how pointers are used. As such, they may not represent the "best" or most efficient way to accomplish a given task. Readers interested in learning more about efficient use of data structures are urged to consult any good text on data structures.

## Creating a Linked List

The following example uses pointers to create and manipulate a linked list. One procedure reads string input from the keyboard and creates a list of pointers to heap variables that have the strings as their values. Another procedure prints the strings, given the pointer to the beginning of the linked list. A third procedure uses a modified "bubble sort" algorithm to reorder the values so the strings are in alphabetical order.

### Creating the List

The following program prompts the user to enter a series of strings from the keyboard. After reading each string, it creates a new heap variable containing a list element—an anonymous structure with two fields; one to hold the string data and one to hold a pointer to the next list element. Any number of strings can be entered. When the user is finished entering strings, the program can be exited by entering a period by itself at the "Enter string:" prompt.

#### Example Code

The source code for this example can be found in the file `ptr_read.pro` in the `examples/doc/language` subdirectory of the IDL distribution. Run the example procedure by entering `ptr_read` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT ptr_read.pro`.

Run the PTR_READ program by entering the following command at the IDL prompt:

```
ptr_read, first
```

Type a string, press Return, and the program prompts for another string. You can enter as many strings as you want. Each time a string is entered, PTR_READ creates a new list element with that string as its value.

For example, you could enter the following three strings (used in the rest of this example):

```
Enter a list of names.
Enter a period (.) to stop list entry.
Enter string: wilma
Enter string: biff
Enter string: cosmo
Enter string: .
```

The following figure shows one way of visualizing the linked list that we've created.



*Table 17-1: One way of visualizing the linked list created by the PTR_READ procedure*

## Printing the Linked List

The next program in our example accepts the pointer to the first element of the linked list and prints all the values in the list in order. To illustrate how the list is linked, we will also print the name of the heap variable that contains each element, and the name of the heap variable in the next field of that element.

### Example Code

The source code for this example can be found in the file ptr_print.pro in the examples/doc/language subdirectory of the IDL distribution. Run the example procedure by entering ptr_print at the IDL command prompt or view the file in an IDL Editor window by entering .EDIT ptr_print.pro.

If we run the PTR_PRINT program with the list generated in the previous example:

```
IDL> ptr_print, first
```

IDL prints:

```
<PtrHeapVar1>, named wilma, has a pointer to: <PtrHeapVar2>
<PtrHeapVar2>, named biff, has a pointer to: <PtrHeapVar3>
<PtrHeapVar3>, named cosmo, has a pointer to: <NullPointer>
```

## A Simple Sorting Routine for the Linked List

The next example program takes a list generated by PTR_READ and moves the values so that they are in alphabetical order. The sorting algorithm used in this program is a variation on the classic "bubble sort". However, instead of starting with

the last element in the list and letting lower values "rise" to the top, this example starts at the top of the list and lets higher ("heavier") values "sink" to the bottom of the list. Note that this is not a very efficient sorting algorithm and is shown as an illustration because of its simplicity. For real sorting applications, use IDL's SORT function.

**Example Code**

The source code for this example can be found in the file `ptr_sort.pro` in the `examples/doc/language` subdirectory of the IDL distribution. Run the example procedure by entering `ptr_sort` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT ptr_sort.pro`.

To run the PTR_SORT routine with the list from our previous examples as input, enter:

```
ptr_sort, first
```

We can see the results of the sorting by calling the PTR_PRINT routine again:

```
ptr_print, first
```

IDL prints:

```
<PtrHeapVar1>, named biff, has a pointer to: <PtrHeapVar2>
<PtrHeapVar2>, named cosmo, has a pointer to: <PtrHeapVar3>
<PtrHeapVar3>, named wilma, has a pointer to: <NullPointer>
```

and we see that now the names are in alphabetical order.

# Example Files—Using Pointers to Create Binary Trees

Two more-complicated example programs demonstrate the use of IDL pointers to create and search a simple tree structure.

**Example Code**

These files, named `idl_tree.pro` and `tree_example.pro`, can be found in the `examples/doc/language` subdirectory of the IDL distribution. Run these example procedures by entering `idl_tree` or `tree_example` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT idl_tree.pro` or `.EDIT tree_example.pro`.

To run the tree examples, enter the following commands at the IDL prompt:

```
; Compile the routines in idl_tree. The example routine calls the
; routines defined in this file.
```

```
.run idl_tree

; Run the tree_example.
tree_example
```

The TREE_EXAMPLE and IDL_TREE routines create a binary tree with ten nodes whose values are structures that contain random values for two fields, "Time" and "Data". The TREE_EXAMPLE routine then prints the tree sorted by both time and data. It then searches for and deletes the nodes containing the fourth and second data values. The resulting 8-node trees are again printed in both time and data order.

A detailed explication of the TREE_EXAMPLE and IDL_TREE routines is beyond the scope of this chapter. Interested users should examine the files, starting with tree_example.pro, to see how the trees are created and searched.

# Chapter 18
# Files and Input/Output

The following topics are covered in this chapter:

# Overview of File Access

IDL provides powerful facilities for file input and output. Few restrictions are imposed on data files by IDL, and there is no unique IDL format. This chapter describes IDL input/output methods and routines and gives examples of programs which read and write data using IDL, C, and FORTRAN.

The first section of this chapter provides a description for how IDL input/output works. It is intentionally brief and is intended to serve only as an introduction. Additional details are covered in the following sections. For the IDL user, perhaps the largest single difference between platforms is input/output. The majority of this chapter covers information that is required in all of the environments IDL supports. Operating system specific information is concentrated in the final sections of this chapter.

## About Opening Files

Before any file input or output can be performed, it is necessary to open a file. This is done using either the OPENR (Open for Reading), OPENW (Open for Writing), or OPENU (Open for Update) procedures. When a file is opened, it is associated with a *Logical Unit Number*, or LUN. All file input and output routines in IDL use the LUN rather than the filename, and most require that the LUN be explicitly specified. Once a file is opened, several input/output routines are available for use. Each routine fills a particular need – the one to use depends on the particular situation.

There are three exceptions to the need to open any file before performing input/output on it. Three files are always open – in fact, the user is not allowed to close them. These files are the *standard input* (usually the keyboard), the *standard output* (usually the IDL log window), and the *standard error output* (usually the terminal screen). These three files are associated with LUNs 0, -1, and -2, respectively. Because these files are always open, there is no need to open them prior to using them for input/output. The READ and PRINT procedures automatically use these files, so basic formatted input/output is extremely simple.

# Simple I/O Examples

It is easy to use input/output using the default input and output files. The IDL command:

```
PRINT, 'Hello World.'
```

causes IDL to print the line:

```
Hello World.
```

on the terminal screen. This happens because PRINT formats its arguments and prints them to LUN -1, which is the standard output file. It is only slightly more complicated to use other files. The following IDL statements show how the above "Hello World" example could be sent to a file named *hello.dat*:

```
;Open LUN 1 for hello.dat with write access.
OPENW, 1, 'hello.dat'

;Do the output operation to the file.
PRINTF, 1, 'Hello World.'

;Close the file.
CLOSE, 1
```

# Routines for Input/Output

See the categories under the functional heading "Input/Output" (*IDL Quick Reference*) for a complete list of available routines.

# Formatted and Unformatted Input/Output

Unformatted Input/Output is the most basic form of input/output. Unformatted input/output transfers the internal binary representation of the data directly between memory and the file. Formatted output converts the internal binary representation of the data to ASCII characters which are written to the output file. Formatted input reads characters from the input file and converts them to internal form. Formatted I/O can be either "Free" format or "Explicit" format, as described below.

## Advantages and Disadvantages of Unformatted I/O

Unformatted input/output is the simplest and most efficient form of input/output. It is usually the most compact way to store data. Unformatted input/output is the least portable form of input/output. Unformatted data files can only be moved easily to and from computers that share the same internal data representation. It should be noted that XDR (eXternal Data Representation) files, described in "Portable Unformatted Input/Output" on page 456, can be used to produce portable binary data. Unformatted input/output is not directly human readable, so you cannot type it out on a terminal screen or edit it with a text editor.

## Advantages and Disadvantages of Formatted I/O

Formatted input/output is very portable. It is a simple process to move formatted data files to various computers, even computers running different operating systems, as long as they all use the ASCII character set. (ASCII is the American Standard Code for Information Interchange. It is the character set used by almost all current computers, with the notable exception of large IBM mainframes.) Formatted files are human readable and can be typed to the terminal screen or edited with a text editor.

However, formatted input/output is more computationally expensive than unformatted input/output because of the need to convert between internal binary data and ASCII text. Formatted data requires more space than unformatted to represent the same information. Inaccuracies can result when converting data between text and the internal representation.

## Free Format I/O

With free format input/output, IDL uses default rules to format the data.

### Advantages and Disadvantages of Free Format I/O

The user is free of the chore of deciding how the data should be formatted. Free format is extremely simple and easy to use. It provides the ability to handle the majority of formatted input/output needs with a minimum of effort. However, the default formats used are not always exactly what is required. In this case, explicit formatting is necessary.

See "Using Free Format Input/Output" on page 401 for more information.

## Explicit Format I/O

Explicit format I/O allows you to specify the exact format for input/output.

### Advantages and Disadvantages of Explicit I/O

Explicit formatting allows a great deal of flexibility in specifying exactly how data will be formatted. Formats are specified using a syntax that is similar to that used in FORTRAN format statements. Scientists and engineers already familiar with FORTRAN will find IDL formats easy to write. Commonly used FORTRAN format codes are supported. In addition, IDL formats have been extended to provide many of the capabilities found in the *scanf ()* and *printf ()* functions commonly found in the C language runtime library.

However, there are some disadvantages to using Explicit I/O. Using explicitly specified formats requires the user to specify more detail—they are, therefore, more complicated to use than free format.

The type of input/output to use in a given situation is usually determined by considering the advantages and disadvantages of each method as they relate to the problem to be solved. Also, when transferring data to or from other programs or systems, the type of input/output is determined by the application. The following suggestions are intended to give a rough idea of the issues involved, though there are always exceptions:

- Images and large data sets are usually stored and manipulated using unformatted input/output in order to minimize processing overhead. The IDL ASSOC function is often the natural way to access such data.

- Data that need to be human readable should be written using formatted input/output.

- Data that need to be portable should be written using formatted input/output. Another option is to use unformatted XDR files by specifying the XDR keyword with the OPEN procedures. This is especially important if moving between computers with markedly different internal binary data formats. XDR is discussed in "Portable Unformatted Input/Output" on page 456.

- Free format input/output is easier to use than explicitly formatted input/output and about as easy as unformatted input/output, so it is often a good choice for small files where there is no strong reason to prefer one method over another.

- Special well-known complex file formats are usually supported directly with special IDL routines (e.g. READ_JPEG for JPEG images).

See "Using Explicitly Formatted Input/Output" on page 406 for more information and examples.

# Opening Files

Before a file can be processed by IDL, it must be opened using one of the procedures described in the following table. All open files are associated with a LUN (Logical Unit Number) within IDL, and all input/output routines refer to files via this number. For example, to open the file named *data.dat* for reading on file unit 1, use the following statement:

```
OPENR, 1, 'data.dat'
```

The OPENR/OPENU/OPENW procedures can be used with certain keywords to modify their normal behavior. Some keywords are generally applicable, while others only have effect under a given operating system. Some operating system specific keywords are allowed (and ignored) under other operating systems in order to facilitate writing portable routines.

| Procedure | Description |
|-----------|-------------|
| OPENR | Opens an existing file for input only. |
| OPENW | Opens a new file for input and output. If the named file already exists, its old contents are overwritten. |
| OPENU | Opens an existing file for input and output. |

*Table 18-1: IDL File Opening Commands*

## Platform-Specific Keywords to the OPEN Procedure

Different computers and operating systems perform input/output in different ways. See "OPENR/OPENU/OPENW" (*IDL Reference Guide*) for keywords to the OPEN procedures that apply under UNIX or Microsoft Windows.

# Closing Files

After work involving the file is complete, it should be closed. Closing a file removes the association between the file and its unit number, thus freeing the unit number for use with a different file. There is usually an operating system-imposed limit on the number of files a user may have open at once. Although this number is large enough that it rarely causes problems, situations can occur where a file must be closed before another file may be opened. In any event, it is good style to only keep needed files open.

There are three ways to close a file:

- Use the CLOSE procedure.
- Use the FREE_LUN procedure on a LUN that has been allocated by GET_LUN.
- Exit IDL. IDL closes all open files when it exits.

Calling the CLOSE procedure is the most common way to close a file unit. For example, to close file unit number 1, use the following statement:

```
CLOSE, 1
```

In addition, if FREE_LUN is called with a file unit number that was previously allocated by GET_LUN, it calls CLOSE before deallocating the file unit. Finally, all open files are automatically closed when IDL exits.

# Understanding (LUNs)

IDL Logical Unit Numbers (LUNs) fall within the range −2 to 128. Some LUNs are reserved for special functions as described below.

## The Standard Input, Output, and Error LUNs

The three LUNs described below have special meanings that are operating system dependent:

### UNIX

Logical Unit Numbers 0, -1, and -2 are tied to *stdin*, *stdout*, and *stderr*, respectively. This means that the normal UNIX file redirection and pipe operations work with IDL. For example, the shell command

```
%idl < idl.inp >& idl.out &
```

will cause IDL to execute in the background, reading its input from the file *idl.inp* and writing its output to the file *idl.out*. Any messages sent to stderr are also sent to *idl.out*.

When using the IDL Workbench, Logical Unit Numbers 0, -1, and -2 are tied to *stdin* (the command line), *stdout* (the log window), and *stderr* (the log window), respectively.

### Windows

Logical Unit Numbers 0, -1, and -2 are tied to *stdin* (the command line), *stdout* (the log window), and *stderr* (the log window), respectively.

These special file units are described in more detail below.

### File Unit 0

This LUN represents the standard input stream, which is usually the keyboard. Therefore, the IDL statement:

```
READ, X
```

is equivalent to the following:

```
READF, 0, X
```

### File Unit -1

This LUN represents the standard output stream, which is usually the terminal screen. Therefore, the IDL statement:

```
PRINT, X
```

is equivalent to the following:

```
PRINTF, -1, X
```

### File Unit -2

This LUN represents the standard error stream, which is usually the terminal screen.

# File Units (1–99)

These are the file units for normal interactive use. When using IDL interactively, the user arbitrarily selects the file units used. The file units from 1 to 99 are available for this use.

# File Units (100–128)

These are the file units managed by the GET_LUN and FREE_LUN procedures. If an IDL procedure or function that uses files is written to explicitly use a given file unit, there is a chance that it will conflict with other routines that use the same unit. It is therefore necessary to avoid explicit file unit numbers when writing IDL procedures and functions. The GET_LUN and FREE_LUN procedures provide a standard mechanism for IDL routines to obtain unique file units. GET_LUN allocates a file unit from a pool of free units in the range 100 to 128. This unit will not be allocated again until it is released by a call to FREE_LUN. Meanwhile, it is available for the exclusive use of the program that allocated it. A typical procedure that needs a file unit might be structured as follows:

```
PRO DEMO
;Get a unique file unit and open the file.
OPENR, UNIT, /GET_LUN

;Body of program goes here.
.
.
.

;Return file unit.
FREE_LUN, UNIT
```

```
;Since the file is still open, FREE_LUN will automatically call
;CLOSE.
END
```

**Note** ─────────────────────────────────────────────

All IDL procedures and functions that open files should use GET_LUN/ FREE_LUN to obtain file units. Furthermore, the file units between 100 and 128 should never be used unless previously allocated by GET_LUN.

─────────────────────────────────────────────────────

# Returning Information About a File Unit

Information about currently open file units is available by using the FILES keyword with the HELP procedure, or using the FSTAT function. If no arguments are provided, information about all currently open user file units (units 1–128) is given. For example, the following command can be used to get information about the three special units (−2, −1, and 0):

```
HELP, /FILES, -2, -1, 0
```

This command results in output similar to the following:

```
Unit   Attributes                        Name
-2      Write, New, Tty, Reserved        <stderr>
-1      Write, New, Tty, Reserved        <stdout>
 0      Read, Tty, Reserved              <stdin>
```

See "HELP" (*IDL Reference Guide*) for details.

## Using FSTAT

The FSTAT function can be used to retrieve information about a file that is currently open (that is, for which there is an IDL Logical Unit Number available). It returns a structure expression of type FSTAT or FSTAT64 containing information about the file. For example, to get detailed information about the standard input, use the following command:

```
HELP, /STRUCTURES, FSTAT(0)
```

This displays the following information:

```
** Structure FSTAT, 17 tags, length=64:
UNIT            LONG         0
NAME            STRING      '<stdin>'
OPEN            BYTE         1
ISATTY          BYTE         0
ISAGUI          BYTE         1
INTERACTIVE     BYTE         1
XDR             BYTE         0
COMPRESS        BYTE         0
READ            BYTE         1
wWRITE           BYTE          0
ATIME           LONG64       0
CTIME           LONG64       0
MTIME           LONG64       0
TRANSFER_COUNT  LONG         0
CUR_PTR         LONG         0
```

```
SIZE              LONG          0
REC_LEN           LONG          0
```

On some platforms, IDL can support files that are longer than 2^31-1 bytes in length. If FSTAT is applied to such a file, it returns an expression of type FSTAT64 instead of the FSTAT structure shown above. FSTAT64 differs from FSTAT only in that the TRANSFER_COUNT, CUR_PTR, SIZE, and REC_LEN fields are signed 64-bit integers (type LONG64) in order to be able to represent the larger sizes.

The fields of the FSTAT and FSTAT64 structures provide various information about the file, such as the size of the file, and the dates of last access, creation, and last modification. For more information on the fields of the FSTAT and FSTAT64 structures, see "FSTAT" (*IDL Reference Guide*).

## An Example Using FSTAT

The following IDL function can be used to read single-precision, floating-point data from a stream file into a vector when the number of elements in the file is not known. It uses the FSTAT function to get the size of the file in bytes and divides by four (the size of a single-precision, floating-point value) to determine the number of values.

```
;READ_DATA reads all the floating point values from a stream file
;and returns the result as a floating-point vector.
FUNCTION READ_DATA, file

;Get a unique file unit and open the data file.
OPENR, /GET_LUN, unit, file

;Get file status.
status = FSTAT(unit)

;Make an array to hold the input data. The SIZE field of status
;gives the number of bytes in the file, and single-precision,
;floating-point values are four bytes each.
data = FLTARR(status.size / 4)

;Read the data.
READU, unit, data

;Deallocate the file unit. The file also will be closed.
FREE_LUN, unit

RETURN, data

END
```

Assuming that a file named `data.dat` exists and contains 10 floating-point values, the READ_DATA function could be used as follows:

```
;Read floating-point values from data.dat.
A = READ_DATA('data.dat')

;Show the result.
HELP, A
```

The following output is produced:

```
A               FLOAT     = Array(10)
```

# File Unit Manipulations

The following sections describe common tasks when working with file units.

## Flushing File Units

For efficiency, IDL buffers its input/output in memory. Therefore, when data are output, there is a window of time during which data are in memory and have not been actually placed into the file. Normally, this behavior is transparent to the user (except for the improved performance). The FLUSH routine exists for those rare occasions where a program needs to be certain that the data has actually been written to the file immediately. For example, use the statement,

```
FLUSH, 1
```

to flush file unit one.

See "FLUSH" (*IDL Reference Guide*) for details.

## Positioning File Pointers

Each open file unit has a current file pointer associated with it. This file pointer indicates the position in the file at which the next input/output operation will take place. The file position is specified as the number of bytes from the start of the file. The first position in the file is position zero. The following statement will rewind file unit 1 to its start:

```
POINT_LUN, 1, 0
```

The following sequence of statements will position it at the end of the file:

```
tmp = FSTAT(1)
POINT_LUN, 1, tmp.size
```

POINT_LUN has the following operating-system specific behavior:

- **UNIX:** the current file pointer can be positioned arbitrarily – moving to a position beyond the current end-of-file causes the file to grow out to that point. The gap created is filled with zeroes.

- **Windows:** the current file pointer can be positioned arbitrarily – moving to a position beyond the current end-of-file causes the file to grow out to that point. Unlike UNIX, the gap created is filled with arbitrary data instead of zeroes.

See "POINT_LUN" (*IDL Reference Guide*) for details.

# Testing for End-Of-File

The EOF function is used to test a file unit to see if it is currently positioned at the end of the file. It returns true (1) if the end-of-file condition is true and false (0) otherwise.

For example, to read the contents of a file and print it on the screen, use the following statements:

```
;Open file demo.doc for reading.
OPENR, 1, 'demo.doc'

;Create a variable of type string.
LINE = ''

;Read and print each line until the end of the file is encountered.
WHILE(~ EOF(1)) DO BEGIN READF,1,LINE & PRINT,LINE & END

;Done with the file.
CLOSE, 1
```

See "EOF" (*IDL Reference Guide*) for details.

# Reading and Writing Very Large Files

IDL on all platforms is able to read and write data from files up to $2^{31}$-1 bytes in length. On some platforms, it is also able to read and write data from files longer than this limit.

To see if IDL on your platform supports large files, use the following:

```
PRINT, !VERSION.FILE_OFFSET_BITS
```

If IDL prints the number 64, the platform supports large files. For more information, see "!VERSION" (*IDL Reference Guide*).

**Warning** —————————————————————————————————

Macintosh systems that use the UNIX File System (UFS) rather than the default Mac OS Extended Filesystem (HFS+) will not be able to access large files, even though IDL itself will report the ability to do so. This is a limitation of the file system, not of IDL.

—————————————————————————————————————————————

When reading and writing to files smaller than this limit, there is no difference in behavior between the platforms that can and those that cannot handle larger files. IDL uses longword integers for file position arguments (e.g. POINT_LUN, FSTAT) and keywords, as before. However, when dealing with files that exceed this limit, IDL uses signed 64-bit integers in order to be able to properly represent the offset. Consider the following example:

```
;Open the file
OPENW, 1, 'test.dat'

;Initial position should be 0.
POINT_LUN, -1, POS

;Print the position and its type.
HELP, POS

;Move the file pointer past the signed 32-bit boundary.
POINT_LUN, 1, '000000ffffffffff'x

;The position is now too large to represent as a longword.
POINT_LUN, -1, POS

;Print the position and its type.
HELP, POS

CLOSE, 1
```

Executing these statements results in the following output:

```
POS             LONG    =                 0
POS             LONG64  =     1099511627775
```

Initially, the file position is 0, which fits easily into a 32-bit integer. Once the file position exceeds the range of a signed 32-bit number, IDL automatically shifts to the 64-bit integer type.

# Limitations of Large File Support

There are limitations on IDL's support for very large files that must be understood by the IDL programmer:

* On any platform, the amount of data that IDL can transfer in a single operation is limited by the amount of memory it can allocate. On most platforms, IDL is a 32-bit program, and as such, can theoretically address up to $2^{31}-1$ bytes of memory (approximately 2.3GB). On these 32-bit platforms, reading, writing, and processing data larger than this limit must be done in multiple operations. Most systems do not have 2.3 GB of memory available, and other programs running on the system also compete for the same memory, so the actual memory available is likely to be considerably smaller.

   To see if your platform is 32- or 64-bit, use the following:

   ```
   PRINT, !VERSION.MEMORY_BITS
   ```

   IF "32" is returned, your platform is 32-bit. If "64" is returned, your platform is 64-bit. For more information, see "!VERSION" (*IDL Reference Guide*).

* The ability to read or write to very large files is constrained by the ability of the underlying file system to support such files. Many platforms can only support large files on certain file systems. For example, many platforms will be unable to support these operations on NFS mounted file systems because NFS version 3 and later must be in use on both client and server. Some platforms can only support such operations on special large file systems, and only if they are mounted using the appropriate mount options. Consult your system documentation to determine the limitations present on your system and the procedures for supporting very large file.

# Using Free Format Input/Output

Use of formatted data is most appropriate when the data must be in human readable form, such as when it is to be prepared or modified with a text editor. Formatted data also are highly portable between various computers and operating systems.

In addition to the PRINT, PRINTF, READ, and READF routines already discussed, the STRING function can be used to generate formatted output that is sent to a string variable instead of a file. The READS procedure can be used to read formatted input from a string variable.

The exact format of the character data may be specified to these routines by providing a format string via the FORMAT keyword. If no format string is given, default formats for each type of data are applied. This method of formatted input/output is called free format. Free format input/output is suitable for most applications involving formatted data. It is designed to provide input/output abilities with a minimum of programming.

## Structures and Free Format Input/Output

IDL structures present a special problem for default formatted input and output. The default format for displaying structure data is to surround the structure with curly braces ({}). For example, if you define an anonymous structure:

```
struct = { A:2, B:3, C:'A String' }
```

and then use default formatted output via the PRINT command:

```
PRINT, struct
```

IDL prints:

```
{       2       3 A String}
```

You might suppose that default formatted input would recognize that the curly braces are part of the formatting and ignore them. This is not the case, however. By default, to read the third field in the structure (the string field) IDL will read from the "A" to the end of the line, including the closing brace.

This behavior, while unsymmetric, seems to be the best choice for default behavior—displaying the result of the PRINT statement on the computer screen. We recommend that you use explicitly formatted input/output when reading and writing structures to disk files, so as not to have to explicitly code around the possibility that your structure may include strings.

# Free Format Input

The following rules are used by IDL to perform free format input:

1.  Input is performed on scalar variables. Array and structure variables are treated as collections of scalar variables. For example,

    ```
    A = INTARR(5)
    READ, A
    ```

    causes IDL to read five separate values to fill each element of the variable A.

2.  If the current input line is empty and there are variables left requiring input, read another line.

3.  If the current input line is not empty but there are no variables left requiring input, the remainder of the line is ignored.

4.  Input data must be separated by commas or white space (tabs, spaces, or new lines).

5.  When reading into a variable of type string, all characters remaining in the current input line are placed into the string.

6.  When reading into numeric variables, every effort is made to convert the input into a value of the expected type. Decimal points are optional and exponential (scientific) notation is allowed. If a floating-point datum is provided for an integer variable, the value is truncated.

7.  When reading into a variable of complex type, the real and imaginary parts are separated by a comma and surrounded by parentheses. If only a single value is provided, it is taken as the real part of the variable, and the imaginary part is set to zero. For example:

    ```
    ;Create a complex variable.
    A = COMPLEX(0)

    ;IDL prompts for input with a colon:
    READ, A

    ;The user enters "(3,4)" and A is set to COMPLEX(3, 4).
    :(3, 4)

    ;IDL prompts for input with a colon:
    READ, A

    ;The user enters "50" and A is set to COMPLEX(50, 0).
    :50
    ```

# Free Format Output

The following rules are used by IDL to perform free format output:

1. The format used to output numeric data is determined by the data type. The formats used are summarized in the table below. The formats are specified in the FORTRAN-like style used by IDL for explicitly formatted input/output.

| Data Type | Format |
|---|---|
| Byte | I4 |
| Int, UInt | I8 |
| Long, ULong | I12 |
| Float | G13.6 |
| Long64, ULong64 | I22 |
| Double | G16.8 |
| Complex | '(', G13.6, ',', G13.6, ')' |
| Double-precision Complex | '(', G16.8, ',', G16.8, ')' |
| String | Output full string on current line. |

*Table 18-2: Formats Used for Free-Format Output*

2. The current output line is filled with characters until one of the following happens:

   A. There is no more data to output.

   B. The output line is full. When output is to a file, the default line width is 80 columns (you can override this default by setting the WIDTH keyword to the OPEN procedure). When the output is to the standard output, IDL uses the current width of your tty or command log window.

   C. An entire row is output in the case of multidimensional arrays.

3. When outputting a structure variable, its contents are bracketed with "{" and "}" characters.

# Example: Free Format Input/Output

IDL free format input/output is extremely easy to use. The following IDL statements demonstrate how to read into a complicated structure variable and then print the results:

```
;Create a structure named "types" that contains seven of the basic
;IDL data types, as well as a floating-point array.
A = {TYPES, A:0B, B:0, C:0L, D:1.0, E:1D, $
  F:COMPLEX(0), G: 'string', E:FLTARR(5)}

;Read free-formatted data from input
READ, A

;IDL prompts for input with a colon. We enter values for the first
;six numeric fields of A and the string.
: 1 2 3 4 5 (6,7) EIGHT
```

Notice that the complex value was specified as (6, 7). If the parentheses had been omitted, the complex field of A would have received the value COMPLEX(6, 0), and the 7 would have been input for the next field. When reading into a string variable, IDL starts from the current point in the input and continues to the end of the line. Thus, we do not enter values intended for the rest of the structure on this line.

```
;There are still fields of A that have not received data, so IDL
;prompts for another line of input.
: 9 10 11 12 13

;Show the result.
PRINT, A
```

Executing these statements results in the following output:

```
{    1        2            3        4.00000        5.0000000
(      6.00000,      7.00000)  eight
         9.00000     10.0000      11.0000       12.0000        13.0000
}
```

When producing the output, IDL uses default rules for formatting the values and attempts to place as many items as possible onto each line. Because the variable A is a structure, braces {} are placed around the output. As noted above, when IDL reads strings it continues to the end of the line. For this reason, it is usually convenient to place string variables at the end of the list of variables to be input.

For example, if S is a string variable and I is an integer:

```
;Read into the string first.
READ, S, I

;IDL prompts for input. We enter a string value followed by an
;integer.
: Hello World 34

;The entire previous line was placed into the string variable S,
;and I still requires input. IDL prompts for another line.
: 34
```

# Using Explicitly Formatted Input/Output

The FORMAT keyword can be used with the formatted input/output routines to explicitly specify the appearance of the data. The standard syntax of IDL format strings is similar to that used in FORTRAN; a C printf()-style syntax is also supported, as described in "C printf-Style Quoted String Format Code" on page 437.

**Note** ──────────────────────────────────────────────

IDL uses the standard I/O function `sprintf` to do its formatting. Different platforms implement this function in different ways, which may lead to slight inconsistencies in the appearance of the output.

─────────────────────────────────────────────────────

The format string specifies the format in which data is to be transferred as well as the data conversion required to achieve that format. The format specification strings supplied by the FORMAT keyword have the form:

```
FORMAT = '(q₁f₁s₁f₂s₂ ... fₙqₙ)'
```

where *q*, *f*, and *s* are described below.

## Record Terminators

*q* is zero or more slash (/) record terminators. On output, each record terminator causes the output to move to a new line. On input, each record terminator causes the next line of input to be read.

## Format Codes

*f* is a format code. Some format codes specify how data should be transferred while others control some other function related to how input/output is handled. The code *f* can also be a nested format specification enclosed in parentheses. This is called a *group specification* and has the following form:

```
...[n](q₁f₁s₁f₂s₂ ... fₙqₙ) ...
```

A group specification consists of an optional repeat count `n` followed by a format specification enclosed in parentheses. Use of group specifications allows more compact format specifications to be written. For example, the format specification:

```
FORMAT = '("Result: ", "<",I5,">", "<",I5,">")'
```

can be written more concisely using a group specification:

```
FORMAT = '("Result: ", 2("<",I5,">"))'
```

If the repeat count is 1 or is not given, the parentheses serve only to group format codes for use in format reversion (discussed in the next section). Format codes and their syntax are described in detail in "Format Codes" on page 411.

### Field Separators

*s* is a field separator. A field separator consists of one or more commas (,) and/or slash record terminators (/). The only restriction is that two commas cannot occur side-by-side.

The arguments provided in a call to a formatted input/output routine are called the *argument list*. The argument list specifies the data to be moved between memory and the file. All data are handled in terms of basic IDL components. Thus, an array is considered to be a collection of scalar data elements, and a structure is processed in terms of its basic components. Complex scalar values are treated as two floating-point values.

# Rules for Explicitly Formatted Input/Output

IDL uses the following rules to process explicitly formatted input/output:

1. Traverse the format string from left to right, processing each record terminator and format code until an error occurs or no data is left in the argument list. The comma field separator serves no purpose except to delimit the format codes.

2. It is an error to specify an argument list with a format string that does not contain a format code that transfers data to or from the argument list because an infinite loop would result.

3. When a slash record terminator (/) is encountered, the current record is completed, and a new one is started. For output, this means that a new line is started. For input, it means that the rest of the current input record is ignored, and the next input record is read.

4.  When a format code that does not transfer data to or from the argument list is encountered, process it according to its meaning. The format codes that do not transfer data to or from the argument list are summarized here.

| Code | Action |
|---|---|
| Quoted String | On output, the contents of the string are written out. On input, quoted strings are ignored. |
| : | The colon format code in a format string terminates format processing if no more items remain in the argument list. It has no effect if data still remains on the list. |
| $ | On output, if a $ format code is placed anywhere in the format string, the new line implied by the closing parenthesis of the format string is suppressed. On input, the $ format code is ignored. |
| *n*H | FORTRAN-style Hollerith string. Hollerith strings are treated exactly like quoted strings. |
| *n*X | Skips *n* character positions. |
| T*n* | Tab. Sets the character position of the next item to the *n*-th position in the current record. |
| TL*n* | Tab Left. Specifies that the next character to be transferred to or from the current record is the *n*-th character to the left of the current position. |
| TR*n* | Tab Right. Specifies that the next character to be transferred to or from the current record is the *n*-th character to the right of the current position. |

*Table 18-3: Format Codes That Do Not Transfer Data*

5.  When a format code that transfers data to or from the argument list is encountered, it is matched up with the next datum in the argument list. The format codes that transfer data to or from the argument list are summarized in the following table.

| Code | Action |
|------|--------|
| A | Transfer character data. |
| B | Transfer binary data. |
| C() | Transfer calendar (Julian date and/or time) data. |
| D | Transfer double-precision, floating-point data. |
| E | Transfer floating-point data using scientific (exponential) notation. |
| F | Transfer floating-point data. |
| G | Use F or E format depending on the magnitude of the value being processed. |
| I | Transfer integer data. |
| O | Transfer octal data. |
| Q | Obtain the number of characters in the input record remaining to be transferred during a read operation. In an output statement, the Q format code has no effect except that the corresponding input/output list element is skipped. |
| Z | Transfer Hexadecimal data. |

*Table 18-4: Format Codes That Transfer Data*

6. On input, read data from the file and format it according to the format code. If the data type of the input data does not agree with the data type of the variable that is to receive the result, do the type conversion if possible; otherwise, issue a type conversion error and stop.

7. On output, write the data according to the format code. If the data type does not agree with the format code, do the type conversion prior to doing the output if possible. If the type conversion is not possible, issue a type conversion error and stop.

8.   If the last closing parenthesis of the format string is reached and there are no data left on the argument list, then format processing terminates. If, however, there are still data to be processed on the argument list, then part or all of the format specification is reused. This process is called *format reversion*.

# Format Reversion

In format reversion, the current record is terminated, a new one is initiated, and format control reverts to the group repeat specification whose opening parenthesis matches the next-to-last closing parenthesis of the format string. If the format does not contain a group repeat specification, format control returns to the initial opening parenthesis of the format string. For example, the IDL command:

```
PRINT, FORMAT = '("The values are: ", 2("<", I1, ">"))', $
   INDGEN(6)
```

results in the output

```
The values are: <0><1>
<2><3>
<4><5>
```

The process involved in generating this output is as follows:

1.   Output the string "The values are: ".

2.   Process the group specification and output the first two values. The end of the format specification is encountered, so end the output record. Data are remaining, so move back to the group specification
```
        2("<", I1, ">")
```
by format reversion.

3.   Repeat Step 2 until no data remain. End the output record. Format processing is complete.

# Format Codes

Format codes specify either how data should be transferred or how input/output is handled.

## Syntax of Format Codes

The syntax of an IDL format code is:

```
[n]FC[+][-][width]
```

Where:

| | |
|---|---|
| *n* | is an optional repeat count ($1 \leq n$) specifying the number of times the format code should be processed. If *n* is not specified, a repeat count of one is used. |
| *FC* | is the format code. See "Available Format Codes", below. |
| + | is an optional flag that specifies that positive numbers should be output with a "+" prefix. The "+" flag is only valid for numeric format codes. Normally, negative numbers are output with a "-" prefix and positive numbers have no sign prefix. Non-decimal numeric codes (B, O, and Z) allow the specification of the "+" flag, but ignore it. |
| - | is an optional flag that specifies that string or numeric values should be output with the text left-justified. Normally, output is right-justified. |
| *width* | is an optional width specification. Width specifications and default values are format-code specific, and are described in detail along with the format code. |
| | See "Padding and Natural Width Formatting", below, for additional information on how output values are formatted based on the *width* parameter. |

## Padding and Natural Width Formatting

The value being formatted may be shorter than the output width specified by the
width parameter. When this happens, IDL will adjust either the contents of the output
value or the width of the field, using the following mechanisms:

### Whitespace Padding

By default, if the value being formatted uses fewer characters than specified by the
*width* parameter, IDL pads the value with whitespace characters on the left to create a
string of the specified width. For example, the following IDL statement

```
PRINT, FORMAT='(I12)', 300
```

produces the following output:

*bbbbbbbbb*300

where *b* represents a space character.

### Zero Padding

For numeric format codes, if the first digit of the *width* parameter is a zero, IDL will
pad the value with zeroes rather than blanks. For example:

```
PRINT, FORMAT='(I08)', 300
```

produces the following output:

00000300

When padding values with zeroes, note the following:

1. If you specify the "-" flag to left-justify the output, specifying a leading zero in
   the *width* parameter has no effect, since there are no unused spaces to the left
   of the output value.

2. If you specify an explicit minimum width value (via the *m* width parameter)
   for an integer format code, specifying a leading zero in the *width* parameter has
   no effect, since the output value is already padded with zeroes on the left to
   create an output value of the specified minimum width.

### Natural Width Formatting

If the numeral zero is specified for the *width* parameter, IDL uses the "natural" width
for the value. The value is read or output using a default format without any leading
or trailing whitespace, in the style of the standard C library printf() function.

Using a value of zero for the *width* parameter is useful when reading tables of data in which individual elements may be of varying lengths. For example, if your data reside in tables of the following form:

```
26.01 92.555 344.2
101.0 6.123 99.845
23.723 200.02 141.93
```

Setting the format to:

```
FORMAT = '(3F0)'
```

ensures that the correct number of digits are read or output for each element.

# Available Format Codes

IDL supports the following format codes:

| Format Code | Description |
| --- | --- |
| A Format Code (page 415) | Transfers character values |
| : Format Code (page 416) | Terminates processing |
| $ Format Code (page 417) | Suppresses newlines in output |
| F, D, E, and G Format Codes (page 418) | Transfer floating-point values |
| B, I, O, and Z Format Codes (page 421) | Transfer integer values |
| Q Format Code (page 424) | Returns the number of characters that remain to be transferred during a read operation |
| Quoted String and H Format Codes (page 425) | Output string values directly |
| T Format Code (page 426) | Specifies the absolute position within a record |
| TL Format Code (page 427) | Moves the position with a record to the left |
| TR and X Format Codes (page 428) | Move the position within a record to the right |

*Table 18-5: Format Codes*

| Format Code | Description |
|---|---|
| C() Format Code (page 429) | Transfers calendar data |
| C printf-Style Quoted String Format Code (page 437) | Provides an alternative syntax for specifying the format of an output string |

*Table 18-5: Format Codes (Continued)*

# Format Code Examples

For examples using different format codes, see:

# A Format Code

The A format code transfers character data.

The syntax is:

```
[n]A[-][w]
```

where the parameters "*n*" and "-" are as described in "Syntax of Format Codes" on page 411 and the width specification is as follows:

| | |
|---|---|
| *w* | is an optional width ($0 \leq w$) specifying the number of characters to be transferred. If *w* is not specified, the entire string is transferred. On output, if *w* is greater than the length of the string, the string is right justified. On input, IDL strings have dynamic length, so *w* specifies the resulting length of input string variables. See "Padding and Natural Width Formatting" on page 412 for additional details on the output width of a formatted value. |

For example, the IDL statement,

```
PRINT, FORMAT = '(A6)', '123456789'
```

generates the following output:

```
123456
```

# : Format Code

The colon format code terminates format processing if there are no more data remaining in the argument list.

The syntax is:

    :

For example, the IDL statement,

    PRINT, FORMAT = '(6(I1, :, ", "))', INDGEN(6)

will output the following comma-separated list of integer values:

    0, 1, 2, 3, 4, 5

The use of the colon format code prevented a comma from being output following the final item in the argument list.

# $ Format Code

When IDL completes output format processing, it normally outputs a newline to terminate the output operation. However, if a "$" format code is found in the format specification, this default newline is not output. The "$" format code is only used on output; it is ignored during input formatting.

The syntax is:

```
$
```

One use for the "$" format code is in prompting for user input in programs that run in a tty rather than in the graphical IDL Workbench. For example, the following simple program show the difference between strings formatted with and without the "$" format code. The first PRINT statement prompts the user for input without forcing the user's response to appear on a separate line from the prompt; the second PRINT statement makes the user enter the response on a separate line.

```
IDL> .run
- PRO format_test
- name=''
- age=0
- PRINT, FORMAT='($, "Enter name")'
- READ, name
- PRINT, FORMAT='("Enter age")'
- READ, age
- PRINT, FORMAT='("You are ", I0, " years old, ", A0)', age, name
- END
% Compiled module: FORMAT_TEST.
```

Running the procedure looks like this:

```
IDL> format_test
Enter name: Pat
Enter age
: 29
You are 29 years old, Pat
IDL>
```

where the values in italics were entered by the user in response to the prompts.

# F, D, E, and G Format Codes

The F, D, E, and G format codes are used to transfer floating-point values between memory and the specified file.

The syntax is:

```
[n]F[+][-][w][.d]
[n]D[+][-][w][.d]
[n]E[+][-][w][.d][Ee]
[n]G[+][-][w][.d][Ee]
```

where the parameters "*n*", "+", and "-" are as described in "Syntax of Format Codes" on page 411 and the width specification is as follows:

| | |
|---|---|
| *w* | is an optional width specification ($0 \leq w \leq 255$). The variable *w* specifies the number of digits to be transferred. See "Padding and Natural Width Formatting" on page 412 for additional details on the output width of a formatted value. |
| *d* | is an optional width specification ($1 \leq d < w$). For the F, D, and E format codes, *d* specifies the number of positions after the decimal point. For the G format code, *d* specifies the number of significant digits displayed. |
| *e* | is an optional width ($1 \leq e \leq 255$) specifying the width of exponent part of the field. IDL ignores this value—it is allowed for compatibility with FORTRAN. |

On input, the F, D, E, and G format codes all transfer *w* characters from the external field and assign them as a real value to the corresponding input/output argument list datum.

The F and D format codes are used to output values using fixed-point notation. The value is rounded to *d* decimal positions and right-justified into an external field that is *w* characters wide. The value of *w* must be large enough to include a minus sign when necessary, at least one digit to the left of the decimal point, the decimal point, and *d* digits to the right of the decimal point. The code D is identical to F (except for its default values for *w* and *d*) and exists in IDL primarily for compatibility with FORTRAN.

The E format code is used for scientific (exponential) notation. The value is rounded to *d* decimal positions and right-justified into an external field that is *w* characters wide. The value of *w* must be large enough to include a minus sign when necessary, at least one digit to the left of the decimal point, the decimal point, *d* digits to the right of the decimal point, a plus or minus sign for the exponent, the character "e" or "E", and at least two characters for the exponent.

**Note**

IDL uses the standard C library function `snprintf()` to format numbers and their exponents. As a result, different platforms may print different numbers of exponent digits.

The G format code uses the F output style when reasonable and E for other values, but displays exactly *d* significant digits rather than *d* digits following the decimal point.

### Overflow

On output, if the field provided is not wide enough, it is filled with asterisks (`*`) to indicate the overflow condition.

### Default Values of the w, d, and e Parameters

If *w*, *d*, or *e* are omitted, the values specified in the following table are used.

| Data Type | *w* | *d* | *e* |
|---|---|---|---|
| Float, Complex | 15 | 7 | 2 (3 for Windows) |
| Double | 25 | 16 | 2 (3 for Windows) |
| All Other Types | 25 | 16 | 2 (3 for Windows) |

*Table 18-6: Floating Format Defaults*

### Format Code Examples

The following table shows the results of the application of various format codes to given data values. Note that normally, the case of the format code is ignored by IDL.

However, the case of the E and G format codes determines the case used to output the exponent in scientific notation.

| Format | Internal Value | Formatted Output |
|--------|---------------|------------------|
| F | 100.0 | *bbbb*100.0000000 |
| F | 100.0D | *bbbbb*100.0000000000000000 |
| F10.0 | 100.0 | *bbbbbb*100. |
| F10.1 | 100.0 | *bbbbb*100.0 |
| F10.4 | 100.0 | *bb*100.0000 |
| F2.1 | 100.0 | ** |
| e11.4 | 100.0 | *b*1.0000e+02<br>1.0000e+002 (Windows)<br>Note that "e10.4" displays "**********" under Windows because the extra "0" added after the "e" makes the string longer than 10 characters. |
| E11.4 | 100.0 | *b*1.0000E+02<br>1.0000E+002 (Windows) |
| g10.4 | 100.0 | *bbbbb*100.0 |
| g10.4 | 10000000.0 | *b*1.000e+07<br>1.000e+007 (Windows) |
| G10.4 | 10000000.0 | *b*1.000E+07<br>1.000E+007 (Windows) |

*Table 18-7: Floating-Point Output Examples*
*("b" represents a blank space)*

# B, I, O, and Z Format Codes

The B, I, O, and Z format codes are used to transfer integer values to and from the specified file. The B format code is used to output binary values, I is used for decimal values, O is used for octal values, and Z is used for hexadecimal values.

The syntax is:

```
[n]B[-][w][.m]
[n]I[+][-][w][.m]
[n]O[-][w][.m]
[n]Z[-][w][.m]
```

where the parameters "*n*", "+", and "-" are as described in "Syntax of Format Codes" on page 411 and the width specification is as follows:

| | |
|---|---|
| *w* | is an optional width specification ($0 \leq w \leq 255$). The variable *w* specifies the number of digits to be transferred. See "Padding and Natural Width Formatting" on page 412 for additional details on the output width of a formatted value. |
| *m* | is an optional minimum number ($1 \leq m \leq 255$) of nonblank digits to be shown on output. The field is zero-filled on the left if necessary. If *m* is omitted or zero, the output is padded with blanks to achieve the specified width. <br><br>**Note -** The *m* parameter is ignored if *w* is zero. |

### Overflow

On output, if the field provided is not wide enough, it is filled with asterisks (*) to indicate the overflow condition.

**Default Values of the w Parameter**

The default values used by the I, O, and Z format codes if *w* is omitted are specified in the following table:

| Data Type | w |
|-----------|---|
| Byte, Int, UInt | 7 |
| Long, ULong, Float | 12 |
| Long64, ULong64 | 22 |
| Double | 23 |
| All Other Types | 12 |

*Table 18-8: Integer Format Defaults*
*(I, O, and Z format codes)*

The default values used by the B format code if *w* is omitted are specified in the following table:

| Data Type | w |
|-----------|---|
| Byte | 8 |
| Int, UInt | 16 |
| Long, ULong | 32 |
| Long64, ULong64 | 64 |
| All Other Types | 32 |

*Table 18-9: Integer Format Defaults*
*(B format code)*

### Format Code Examples

The following table shows the results of the application of various format codes to given data values. Note that normally, the case of the format code is ignored by IDL. However, the case of the Z format codes determines the case used to output the hexadecimal digits A-F.

| Format | Internal Value | Formatted Output |
|--------|----------------|------------------|
| B | 3000 | *bbbb*101110111000 |
| B15 | 3000 | *bbb*101110111000 |
| B14.14 | 3000 | 00101110111000 |
| I | 3000 | *bbb*3000 |
| I6.5 | 3000 | *b*03000 |
| I5.6 | 3000 | ***** |
| I2 | 3000 | ** |
| O | 3000 | *bbb*5670 |
| O6.5 | 3000 | *b*05670 |
| O5.6 | 3000 | ***** |
| O2 | 3000 | ** |
| z | 3000 | *bbbb*bb8 |
| Z | 3000 | *bbbb*BB8 |
| Z6.5 | 3000 | *b*00BB8 |
| Z5.6 | 3000 | ***** |
| Z2 | 3000 | ** |

*Table 18-10: Integer Output Examples*
*("b" represents a blank space)*

# Q Format Code

The Q format code returns the number of characters in the input record remaining to be transferred during the current read operation. It is ignored during output formatting.

The syntax is:

```
q
```

Format Q is useful for determining how many characters have been read on a line. For example, the following IDL statements count the number of characters in file *demo.dat*:

```
;Open file for reading.
OPENR, 1, "demo.dat"

;Create a longword integer to keep the count.
N = 0L

;Count the characters.
WHILE(~ EOF(1)) DO BEGIN
  READF, 1, CUR, FORMAT = '(q)' & N = N + CUR
ENDWHILE

;Report the result.
PRINT, FORMAT = '("counted", N, "characters.")'

;Close file.
CLOSE, 1
```

# Quoted String and H Format Codes

On output, any quoted strings or Hollerith constants are sent directly to the output. On input, they are ignored.

The syntax for a quoted string is:

```
"string" or 'string'
```

where *string* is the string to be output.

**Note** ————————————————————————————————————

Quoted strings must be enclosed in either single or double quotation marks; use the type of quotation mark that is *not* used to enclose the entire format string.

————————————————————————————————————————————

For example, the IDL statement,

```
PRINT, FORMAT = '("Value: ", I0)', 23
```

results in the following output:

```
Value: 23
```

Note that it would have been equally correct to use double quotes around the entire format string and single quotes around the quoted string "Value: ".

Another way to specify a quoted string is with a Hollerith constant.

The syntax for a Hollerith constant is:

$$nHc_1c_2\ c_3\ ...\ c_n$$

where

| $n$ | is the number of characters in the constant ($1 \le n \le 255$). |
|-----|------------------------------------------------------------------|
| $c_i$ | is the characters that make up the constant. The number of characters must agree with the value provided for *n*. |

For example, the following IDL statement,

```
PRINT, FORMAT = '(7HValue: , I0)', 23
```

results in the following output:

```
Value: 23
```

See "C printf-Style Quoted String Format Code" on page 437 for an alternate form of the Quoted String Format Code that supports C printf-style capabilities.

# T Format Code

The T format code specifies the absolute position in the current record.

The syntax is:

```
Tn
```

where

| | |
|---|---|
| *n* | is the absolute character position within the record to which the current position should be set ($1 \leq n$). |

The T format code differs from the TL, TR, and X format codes primarily in that it specifies an absolute position rather than an offset from the current position. For example,

```
PRINT, FORMAT = '("First", 20X, "Last", T10, "Middle")'
```

produces the following output:

```
FirstbbbbMiddlebbbbbbbbbbbLast
```

where "*b*" represents a blank space.

# TL Format Code

The TL format code moves the current position in the external record to the left.

The syntax is:

```
TLn
```

where

| | |
|---|---|
| *n* | is the number of characters to move left from the current position ($1 \leq n$). If the value of *n* is greater than the current position, the current position is moved to column one. |

The TL format code is used to move backwards in the current record. It can be used on input to read the same data twice or on output to position the output nonsequentially. For example,

```
PRINT, FORMAT = '("First", 20X, "Last", TL15, "Middle")'
```

produces the following output:

```
FirstbbbbbbbbbbMiddlebbbbbLast
```

where "*b*" represents a blank space.

# TR and X Format Codes

The TR and X format codes move the current position in the record to the right.

The syntax is:

```
TRn
nX
```

where

| | |
|---|---|
| *n* | is the number of characters to skip ($1 \leq n$). On input, *n* characters in the current input record will be passed over. On output, the current output position is moved *n* characters to the right. |

The TR or X format codes can be used to leave whitespace in the output or to skip over unwanted data in the input. For example, either

```
PRINT, FORMAT = '("First", 15X, "Last")'
```

or

```
PRINT, FORMAT = '("First", TR15, "Last")'
```

results in the following output:

```
First\textit{bbbbbbbbbbbbbbb}Last
```

where "*b*" represents a blank space.

These two format codes differ in one way. Using the X format code at the end of an output record will not cause any characters to be written unless it is followed by another format code that causes characters to be output. The TR format code always writes characters in this situation. Thus,

```
PRINT, FORMAT = '("First", 15X)'
```

results in the following output:

```
First
```

whereas

```
PRINT, FORMAT = '("First", TR15)'
```

results in the following output:

```
First\textit{bbbbbbbbbbbbbbb}
```

where "*b*" represents a blank space. The X code does not cause the blanks to be output unless there is additional output following the blanks.

# C() Format Code

The C() format code is used to transfer calendar (Julian date and/or time) data.

The syntax is:

```
[n]C([c0,c1,...,cx])
```

where the parameter "*n*" is as described in "Syntax of Format Codes" on page 411 and:

| | |
|---|---|
| $c_i$ | represents optional calendar format subcodes, or any of the standard format codes that are allowed within a calendar specification, as described below |

If no $c_i$ are provided, the data will be transferred using the standard 24-character system format that includes the day, date, time, and year, as shown in this string:

```
Thu Aug 13 12:01:32 1979
```

For input, this default is equivalent to:

```
C(CDwA, X, CMoA, X, CDI, X, CHI, X, CMI, X, CSI, CYI5)
```

For output, this default is equivalent to:

```
C(CDwA, X, CMoA, X, CDI2.2, X, CHI2.2, ":", CMI2.2, ":", CSI2.2,
CYI5)
```

**Note** ────────────────────────────────────────────────────────

The C() format code represents an atomic data transfer. Nesting within the parentheses is not allowed.

────────────────────────────────────────────────────────────────

**Note** ────────────────────────────────────────────────────────

For input using the calendar format codes, a small offset is added to each Julian date to eliminate roundoff errors when calculating the day fraction from hours, minutes, and seconds. This offset is given by the larger of EPS and EPS\**Julian*, where *Julian* is the integer portion of the Julian date, and EPS is the EPS field from MACHAR. For typical Julian dates, this offset is approximately $6 \times 10^{-10}$ (which corresponds to $5 \times 10^{-5}$ seconds). This offset ensures that if the Julian date is converted back to hour, minute, and second, then the hour, minute, and second will have the same integer values as were originally input.

────────────────────────────────────────────────────────────────

**Note** ───────────────────────────────────────────────────────────────

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

## Calendar Format Subcodes

The following is a list of the subcodes allowed within the parenthesis of the C() format code.

**Note** ───────────────────────────────────────────────────────────────

The calendar format subcodes are based on the A, I, and F format codes, and share the same options. See "Syntax of Format Codes" on page 411 for additional information on the parameters not described explicitly in this section. Note that the default values of the *w* and *d* parameters are different in the calendar format subcodes than in the base A, I, and F format codes.

## CMOA Subcodes

The CMOA subcodes transfers the month portion of a date as a string. The format for an all upper case month string is:

    CMOA[-][w]

The format for a capitalized month string is:

    CMoA[-][w]

The format for an all lower case month string is:

    CmoA[-][w]

where:

| | |
|---|---|
| *w* | is an optional width ($0 \leq w$) specifying the number of characters of the month name to be transferred. If *w* is not specified, three characters will be transferred. See "Padding and Natural Width Formatting" on page 412 for additional details on the output width of a formatted value. |

**Note** ───────────────────────────────────────────────────────────────

The case of the 'M' and 'O' of these subcodes will be ignored on input, or if the MONTHS keyword for the current routine is explicitly set.

## CMOI Subcode

The CMOI subcode transfers the month portion of a date as an integer. The format is as follows:

```
CMOI[+][-][w][.m]
```

where:

| | |
|---|---|
| *w* | is an optional width ($0 \le w \le 255$) specifying the width of the field in characters. The default width is 2. See "Padding and Natural Width Formatting" on page 412 for additional details on the output width of a formatted value. |
| *m* | is an optional minimum number ($1 \le m \le 255$) of nonblank digits to be shown on output. The field is zero-filled on the left if necessary. If *m* is omitted or zero, the output is padded with blanks to achieve the specified width.<br>**Note -** The *m* parameter is ignored if *w* is zero. |

## CDI Subcode

The CDI subcode transfers the day portion of a date as an integer. The format is:

```
CDI[+][-][w][.m]
```

where:

| | |
|---|---|
| *w* | is an optional width ($0 \le w \le 255$) specifying the width of the field in characters. The default width is 2. See "Padding and Natural Width Formatting" on page 412 for additional details on the output width of a formatted value. |
| *m* | is an optional minimum number ($1 \le m \le 255$) of nonblank digits to be shown on output. The field is zero-filled on the left if necessary. If *m* is omitted or zero, the output is padded with blanks to achieve the specified width.<br>**Note -** The *m* parameter is ignored if *w* is zero. |

## CYI Subcode

The CYI subcode transfers the year portion of a date as an integer. The format is as follows:

```
CYI[+][-][w][.m]
```

where:

| | |
|---|---|
| *w* | is an optional width ($0 \leq w \leq 255$) specifying the width of the field in characters. The default width is 4. See "Padding and Natural Width Formatting" on page 412 for additional details on the output width of a formatted value. |
| *m* | is an optional minimum number ($1 \leq m \leq 255$) of nonblank digits to be shown on output. The field is zero-filled on the left if necessary. If *m* is omitted or zero, the output is padded with blanks to achieve the specified width.<br><br>**Note -** The *m* parameter is ignored if *w* is zero. |

## CHI Subcodes

The CHI subcodes transfer the hour portion of a date as an integer. The format for a 24-hour based integer is:

```
CHI[+][-][w][.m]
```

The format for a 12 hour based integer is:

```
ChI[+][-][w][.m]
```

where:

| | |
|---|---|
| *w* | is an optional width ($0 \leq w \leq 255$) specifying the width of the field in characters. The default width is 2. See "Padding and Natural Width Formatting" on page 412 for additional details on the output width of a formatted value. |
| *m* | is an optional minimum number ($1 \leq m \leq 255$) of nonblank digits to be shown on output. The field is zero-filled on the left if necessary. If *m* is omitted or zero, the output is padded with blanks to achieve the specified width.<br><br>**Note -** The *m* parameter is ignored if *w* is zero. |

**Note** ——————————————————————————————————
> For the ChI (12 hour format), the CAPA Subcodes may be used to specify A.M. versus P.M. For CHI (24 hour format), the CAPA subcode is ignored."
————————————————————————————————————————

## CMI Subcode

The CMI subcode transfers the minute portion of a date as an integer. The format is:

    CMI[+][-][w][.m]

where:

| | |
|---|---|
| *w* | is an optional width ($0 \leq w \leq 255$) specifying the width of the field in characters. The default width is 2. See "Padding and Natural Width Formatting" on page 412 for additional details on the output width of a formatted value. |
| *m* | is an optional minimum number ($1 \leq m \leq 255$) of nonblank digits to be shown on output. The field is zero-filled on the left if necessary. If *m* is omitted or zero, the output is padded with blanks to achieve the specified width. <br><br> **Note -** The *m* parameter is ignored if *w* is zero. |

## CSI Subcode

The CSI subcode transfers the seconds portion of a date as an integer. The format is:

    CSI[+][-][w][.m]

where:

| | |
|---|---|
| *w* | is an optional width ($0 \leq w \leq 255$) specifying the width of the field in characters. The default width is 2. See "Padding and Natural Width Formatting" on page 412 for additional details on the output width of a formatted value. |
| *m* | is an optional minimum number ($1 \leq m \leq 255$) of nonblank digits to be shown on output. The field is zero-filled on the left if necessary. If *m* is omitted or zero, the output is padded with blanks to achieve the specified width. <br><br> **Note -** The *m* parameter is ignored if *w* is zero. |

## CSF Subcode

The CSF subcode transfers the seconds portion of a date as a floating-point value.
The format is:

```
CSF[+][-][w][.d]
```

where:

| | |
|---|---|
| *w* | is an optional width specification ($0 \leq w \leq 255$). The variable *w* specifies the number of characters in the external field; the default is 5. See "Padding and Natural Width Formatting" on page 412 for additional details on the output width of a formatted value. |
| *d* | is an optional width specification ($1 \leq d < w$). The variable *d* specifies the number of positions after the decimal point; the default is 2. |

### Overflow

The value of *w* must be large enough to include at least one digit to the left of the
decimal point, the decimal point, and *d* digits to the right of the decimal point. On
output, if the field provided is not wide enough, it is filled with asterisks (*) to
indicate the overflow condition.

## CDWA Subcodes

The CDWA subcodes transfers the day of week portion of a data as a string. The
format for an all upper case day of week string is:

```
CDWA[-][w]
```

The format for a capitalized day of week string is:

```
CDwA[-][w]
```

The format for an all lower case day of week string is:

```
CdwA[-][w]
```

where:

| | |
|---|---|
| *w* | is an optional width ($0 \leq w$), specifying the number of characters of the day of week name to be transferred. If *w* is not specified, three characters will be transferred. See "Padding and Natural Width Formatting" on page 412 for additional details on the output width of a formatted value. |

**Note** ───────────────────────────────────

The case of the 'D' and 'W' of these subcodes will be ignored on input, or if the DAYS_OF_WEEK keyword for the current routine is explicitly set.

───────────────────────────────────────────

## CAPA Subcodes

The CAPA subcodes transfers the A.M. or P.M. portion of a date as a string. The format for an all-uppercase A.M. or P.M. string is:

```
CAPA[-][w]
```

The format for a capitalized A.M. or P.M. string is:

```
CApA[-][w]
```

The format for an all-lowercase A.M. or P.M. string is:

```
CapA[-][w]
```

where:

| | |
|---|---|
| *w* | is an optional width ($0 \leq w$), specifying the number of characters of the A.M. or P.M. string to be transferred. If *w* is not specified, two characters will be transferred. See "Padding and Natural Width Formatting" on page 412 for additional details on the output width of a formatted value. |

**Note** ───────────────────────────────────

The case of the first 'A' and 'P' of these subcodes will be ignored on input, or if the AM_PM keyword for the current routine is explicitly set.

───────────────────────────────────────────

**Note** ───────────────────────────────────

The CAPA subcode is only used if the ChI (12 hour format) subcode is also being used. The CAPA subcode is ignored if the CHI (24 hour format) subcode is being used.

───────────────────────────────────────────

## Standard Format Codes Allowed Within a Calendar Specification

None of these subcodes are allowed outside of a C() format specifier. In addition to the subcodes listed above, only quoted strings, "TL", "TR", and "X" format codes are allowed inside of the C() format specifier.

### Example:

To print the current date in the default format:

```
PRINT, FORMAT='(C())', SYSTIME(/JULIAN)
```

The printed result should look something like:

```
Fri Aug 14 12:34:14 1998
```

### **Example:**

To print the current date as a two-digit month value followed by a slash followed by a two-digit day value:

```
PRINT, FORMAT='(C(CMOI,"/",CDI))',SYSTIME(/JULIAN)
```

The printed result should look something like:

```
8/14
```

### **Example:**

To print the current time in hours, minutes, and floating-point seconds, all zero-filled if necessary, and separated by colons:

```
PRINT, $
    FORMAT='(C(CHI2.2,":",CMI2.2,":",CSF05.2))',SYSTIME(/JULIAN)
```

The printed result should look something like:

```
09:59:07.00
```

Note that to do zero-filling for the floating-point seconds, it is necessary to specify a leading 0 in the width to the CSF format code.

# C printf-Style Quoted String Format Code

IDL's explicitly formatted specifications, which are based on those found in the FORTRAN language, are extremely powerful and capable of specifying almost any desired output. However, they require fairly verbose specifications, even in simple cases. In contrast, the C language (and the many languages influenced by C) have a different style of format specification used by functions such as `printf()` and `snprintf()`. Most programmers are very familiar with such formats. In this style, text and format codes (prefixed by a % character) are intermixed in a single string. User-supplied arguments are substituted into the format in place of the format specifiers. Although less powerful, this style of format is easier to read and write in common simple cases.

IDL supports the use of `printf`-style formats within format specifications, using a special variant of the Quoted String Format Code (discussed in "Quoted String and H Format Codes" on page 425) in which the opening quote starts with a % character (e.g. %" or %' rather than " or '). The presence of this % before the opening quote (with no whitespace between them) tells IDL that this is a `printf`-style quoted string and not a standard quoted string.

As a simple example, consider the following IDL statement that uses normal quoted string format codes:

```
PRINT, FORMAT='("I have ", I0, " monkeys, ", A, ".")', $
   23, 'Scott'
```

Executing this statement yields the output:

```
I have 23 monkeys, Scott.
```

Using a `printf`-style quoted string format code instead, this statement could be written:

```
PRINT, FORMAT='(%"I have %d monkeys, %s.")', 23, 'Scott'
```

These two statements are completely equivalent in their action. In fact, IDL compiles both into an identical internal representation before processing them.

The `printf`-style quoted string format codes can be freely mixed with any other format code, so hybrid formats like the following are allowed:

```
PRINT, $
   FORMAT='(%"I have %d monkeys, %s,", " and ", I0, " parrots.")',$
   23, 'Scott', 5
```

This generates the output:

```
I have 23 monkeys, Scott, and 5 parrots.
```

## Supported "%" Formats

The following table lists the % format codes allowed within a `printf`-style quoted string format code, as well as their correspondence to the standard format codes that do the same thing. In addition to the format codes described in the table, the special sequence %% causes a single % character to be written to the output. This % is treated as a regular character instead of as a format code specifier. Finally, the flags and the width padding options described in "Syntax of Format Codes" on page 411 are also available when using `printf`-style format codes.

| Printf-Style | Normal-Style | Normal Style Described in Section |
|---|---|---|
| %[*w.d*]e or %[*w.d*]E | e[*w.d*] or E[*w.d*] | "F, D, E, and G Format Codes" on page 418 |
| %[*w*]b or %[*w*]B<br>%[*w.m*]b or %[*w.m*]B | B[*w*]<br>B[*w.m*] | "B, I, O, and Z Format Codes" on page 421 |
| %[*w*]d or %[*w*]D<br>%[*w.m*]D or %[*w.m*]D<br>%[*w*]i or %[*w*]I<br>%[*w.m*]i or %[*w.m*]I | I[*w*]<br>I[*w.m*]<br>I[*w*]<br>I[*w.m*] | "B, I, O, and Z Format Codes" on page 421 |
| %[*w*]f or %[*w*]F<br>%[*w.d*]f or %[*w.d*]F | F[*w*]<br>F[*w.d*] | "F, D, E, and G Format Codes" on page 418 |
| %[*w*]g or %[*w*]G<br>%[*w.d*]g or %[*w.d*]G | g[*w*] or G[*w*]<br>g[*w.d*] or G[*w.d*] | "F, D, E, and G Format Codes" on page 418 |
| %[*w*]o or %[*w*]O<br>%[*w.m*]o or %[*w.m*]O | O[*w*]<br>O[*w.m*] | "B, I, O, and Z Format Codes" on page 421 |
| %[*w*]s or %[*w*]S | A[*w*] | "A Format Code" on page 415 |
| %[*w*]x or %[*w*]X<br>%[*w.m*]x or %[*w.m*]X<br>%[*w*]z or %[*w*]Z<br>%[*w.m*]z or %[*w.m*]Z | Z[*w*]<br>Z[*w.m*]<br>Z[*w*]<br>Z[*w.m*] | "B, I, O, and Z Format Codes" on page 421 |

*Table 18-11: Supported "%" Formats*

As indicated in the above table, there is a one to one correspondence between each `printf`-style % format code and one of the normal format codes documented earlier in this chapter. When reading this table, please keep the following considerations in mind:

- The %d (or %D) format is identical to the %i (or %I) format. Note that %D does not correspond to the normal-style D format.

- The *w*, *d*, *m*, and *e* parameters listed as optional parameters (i.e. between the square brackets, [ ]) are the same values documented for the normal-style format codes, and behave identically to them.

- The default value for the *w* parameters for `printf`-style formatting is 0, meaning that `printf`-style output produces "natural" width by default. For example, a %d format code corresponds to a normal format code of I0 (not I, which would use the default value for *w* based on the data type). Similarly, a %e format code corresponds to a normal format code of e0 (not e).

- The E and G format codes allow the following styles for compatibility with FORTRAN:

  ```
  E[w.dEe] or e[w.dEe]
  G[w.dEe] or g[w.dEe]
  ```

  These styles are not available using the `printf`-style format codes. In other words, the following formats are not allowed:

  ```
  %[w.dEe]E or %[w.dEe]e
  %[w.dEe]G or %[w.dEe]g
  ```

- Normal-style format codes allow repetition counts (e.g., 5I0). The `printf`-style format codes do not allow this. Instead, each `printf`-style format code has an implicit repetition count of 1.

- Like normal format codes (but unlike the C language `printf()` function), `printf`-style format codes are allowed to be upper or lower case (e.g. %d and %D mean the same thing). Whether or not case has an influence on the resulting output depends on the specific format code. The specific behavior is the same as with the normal-style version for each code.

## Supported "\" Character Escapes

The C programming language allows "escape sequences" that start with the backslash character, \, to appear within strings. These escapes are used in several ways:

1.  To specify characters that have no printed representation. For example, \n means linefeed, and \r means carriage return.

2.  To remove any special meaning that a character might normally have. For example, \" allows you to create a string containing a double-quote character even though double-quote normally delimits a string. Note that backslash can also be used to escape itself, so "\\" corresponds to a string containing a single backslash character.

3.  To introduce arbitrary characters into a string using octal or hexadecimal notation. For example, if the hexadecimal value b1 represents the ± character in the current font, then the following statement:

    ```
    print, format='(%"I have \xb1%d monkeys")', 5
    ```

    results in the following output:

    ```
    I have ±5 monkeys
    ```

Although IDL does not normally support backslash escapes within strings, the escapes described in the following table are allowed within printf-style quoted string format codes. If a character not specified in this table is preceded by a backslash, the backslash is removed and the character is inserted into the output without any special interpretation. This means that \" puts a single " character into the output and that " does not terminate the string constant. Another useful example is that \% causes a single % character to be placed into the output without starting a format code. Hence, \% and %% mean the same thing: a single % character with no special meaning.

| Escape Sequence | ASCII code |
|---|---|
| \a | BEL (7B) |
| \b | Backspace (8B) |
| \f | Formfeed (12B) |
| \n | Linefeed (10B) |

*Table 18-12: Supported "\" Character Escapes*

| Escape Sequence | ASCII code |
|---|---|
| \r | Carriage Return (13B) |
| \t | Horizontal Tab (9B) |
| \v | Vertical Tab (11B) |
| \ooo | Octal value ooo (Octal value of 1-3 digits) |
| \xhh | Hexadecimal value hh (Hex value of 1-2 digits) |

*Table 18-12: Supported "\" Character Escapes (Continued)*

**Note** ———————————————————————————————————

Case is ignored in escape sequences: either "\n" or "\N" specifies a linefeed character.

## Differences Between C printf() and IDL printf-Style Formats

IDL's printf-style quoted string format code is very similar to a simplified C language printf() format string. However, there are important differences that an experienced C programmer should be aware of:

- The IDL PRINT and PRINTF procedures implicitly add an end-of-line character to the end of the line (unless suppressed by use of the $ format code). Hence, the use of \n at the end of the format string to end the line is neither necessary nor recommended.

- Only the % format sequences listed in the table under "Supported "%" Formats" on page 438 are understood by IDL. Most C printf functions accept more codes than these, but those codes are not necessary in IDL.

  For example, the C printf/scanf functions require the use of the %u format code to indicate an unsigned value, and also use type modifiers (h, l, ll) to indicate the size of the data being processed. IDL uses the type of the arguments being substituted into the format to determine this information. Therefore, the u, h, l, and ll codes are not required in IDL and are not accepted.

- The % and \ sequences in IDL printf-style strings are case-insensitive. C printf is case-sensitive (e.g. \n and \N do not both mean the linefeed character as they do in IDL).

- The C `printf` function allows the use of %n$d notation to specify that arguments should be substituted into the format string in a different order than they are listed. IDL does not support this.

- The C `printf` function allows the use of %*d notation to indicate that the field width will be supplied by the next argument, and the argument following that supplies the actual value. IDL does not support this.

- IDL `printf`-style formats allow %z for hexadecimal output as well as %x. The C `printf()` function does not understand %z. This deviation from the usual implementation is allowed by IDL because IDL programmers are used to treating Z as the hexadecimal format code.

- IDL `printf`-style formats allow %b for binary output. The C `printf()` function does not understand %b.

# Example: Reading Formatted Table Data

IDL explicitly formatted input/output has the power and flexibility to handle almost any kind of formatted data. A common use of explicitly formatted input/output involves reading and writing tables of data. Consider a data file containing employee data records. Each employee has a name (String, 32 columns) and the number of years they have been employed (Integer, 3 columns) on the first line. The next two lines contain each employee's monthly salary for the last twelve months. A sample file named *employee.dat* with this format might look like the following:

```
Bullwinkle                      10
1000.0    9000.97   1100.0                            2000.0
5000.0    3000.0    1000.12   3500.0    6000.0    900.0
Boris                           11
400.0     500.0     1300.10   350.0     745.0     3000.0
200.0     100.0     100.0     50.0      60.0      0.25
Natasha                         10
950.0     1050.0    1350.0    410.0     797.0     200.36
2600.0    2000.0    1500.0    2000.0    1000.0    400.0
Rocky                           11
1000.0    9000.0    1100.0    0.0       0.0       2000.37
5000.0    3000.0    1000.01   3500.0    6000.0    900.12
```

The following IDL statements read data with the above format and produce a summary of the contents of the file:

```
;Open data file for input.
OPENR, 1, 'employee.dat'

;Create variables to hold the name, number of years, and monthly
;salary.
name = '' & years = 0 & salary = FLTARR(12)

;Output a heading for the summary.
PRINT, FORMAT='("Name", 28X, "Years", 4X, "Yearly Salary")'

;Note: The actual dashed line is longer than is shown here.
PRINT, '========'

;Loop over each employee.
WHILE (~ EOF(1)) DO BEGIN

   ;Read the data on the next employee.
   READF, 1, $
   FORMAT = '(A32,I3,2(/,6F10.2))', name, years, salary
```

```
;Output the employee information. Use TOTAL to sum the monthly
;salaries to get the yearly salary.
   PRINT, FORMAT='(A32,I5,5X,F10.2)', name, years, TOTAL(salary)

ENDWHILE

CLOSE, 1
```

The output from executing these statements on *employee.dat* is as follows:

```
Name                             Years   Yearly Salary
======================================================
Bullwinkle                          10       32501.09
Boris                               11        6805.35
Natasha                             10       14257.36
Rocky                               11       32500.50
```

# Example: Reading Records With Multiple Array Elements

Frequently, data are written to files with each record containing single elements of more than one array. One example might be a file consisting of observations of altitude, pressure, temperature, and velocity with each line or record containing a value for each of the four variables. Because IDL has no equivalent of the FORTRAN implied DO list, special procedures must be used to read or write this type of file.

The first approach, which is the simplest, may be used only if all of the variables have the same data type. An array is created with as many columns as there are variables and as many rows as there are elements. The data are read into this array, the array is transposed storing each variable as a row, and each row is extracted and stored into a variable which becomes a vector. For example, the FORTRAN program which writes the data and the IDL program which reads the data are as follows:

## FORTRAN Write:

```
DIMENSION ALT(100),PRES(100),TEMP(100),VELO(100)
OPEN (UNIT = 1, STATUS='NEW', FILE='TEST')
WRITE(1,'(4(1x,g15.5))')
     (ALT(I),PRES(I),TEMP(I),VELO(I),I=1,100)
```

## IDL Read:

```
;Open file for input.
OPENR, 1, 'test'

;Define variable (NVARS by NOBS).
A = FLTARR(4,100)

;Read the data.
READF, 1, A

;Transpose so that columns become rows.
A = TRANSPOSE(A)

;Extract the variables.
ALT = A[*, 0]
PRES = A[*, 1]
TEMP = A[*, 2]
VELO = A[*, 3]
```

Note that this same example may be written without the implied DO list, writing all elements for each variable contiguously and simplifying matters considerably:

## FORTRAN Write:

```
DIMENSION ALT(100),PRES(100),TEMP(100),VELO(100)
OPEN (UNIT = 1, STATUS='NEW', FILE='TEST')
WRITE (1,'(4(1x,G15.5))') ALT,PRES,TEMP,VELO
```

## IDL Read:

```
;Define variables.
ALT = FLTARR(100)
PRES = ALT & TEMP = ALT & VELO = ALT
OPENR, 1, 'test'
READF, 1, ALT, PRES, TEMP, VELO
```

A different approach must be taken when the columns contain different data types or the number of lines or records are not known. This method involves defining the arrays, defining a scalar variable to contain each datum in one record, then writing a loop to read each line into the scalars, and then storing the scalar values into each array. For example, assume that a fifth variable, the name of an observer which is of string type, is added to the variable list. The FORTRAN output routine and IDL input routine are as follows:

## FORTRAN Write:

```
DIMENSION ALT(100),PRES(100),TEMP(100),VELO(100)
CHARACTER * 10 OBS(100)
OPEN (UNIT = 1, STATUS = 'NEW', FILE = 'TEST')
WRITE (1,'(4(1X,G15.5),2X,A)')
   (ALT(I),PRES(I),TEMP(I),VELO(I),OBS(I),I=1,100)
```

## IDL Read:

```
;Access file. Read files containing from 1 to 200 records.
OPENR, 1, 'test'

;Define vector, make it large enough for the biggest case.
ALT = FLTARR(200)

;Define other vectors using the first.
PRES = ALT & TEMP = ALT & VELO = ALT

;Define string array.
OBS = STRARR(200)

;Define scalar string.
```

```
OBSS = ''

;Initialize counter.
I = 0

WHILE ~ EOF(1) DO BEGIN
   ;Read scalars.
   READF, 1, $

   FORMAT = '(4(1X, G15.5), 2X, A10)', $
     ALTS, PRESS, TEMPS, VELOS, OBSS

;Store in each vector.
   ALT[I] = ALTS & PRES[I] = PRESS & TEMP[I] = TEMPS
   VELO[I] = VELOS & OBS[I] = OBSS

   ;Increment counter and check for too many records.
   IF I LT 199 THEN I = I + 1 ELSE STOP, 'Too many records'
ENDWHILE
```

If desired, after the file has been read and the number of observations is known, the arrays may be truncated to the correct length using a series of statements similar to the following:

```
ALT = ALT[0:I-1]
```

The above statement represents a worst case example. Reading is greatly simplified by writing data of the same type contiguously and by knowing the size of the file. One frequently used technique is to write the number of observations into the first record so that when reading the data the size is known.

**Warning** —————————————————————————————————————————

It might be tempting to implement a loop in IDL which reads the data values directly into array elements, using a statement such as the following:

```
FOR I = 0, 99 DO READF, 1, ALT[I], PRES[I], TEMP[I], VELO[I]
```

This statement is *incorrect*. Subscripted elements (including ranges) are temporary expressions passed as values to procedures and functions (READF in this example). Parameters passed by value do not pass results back to the caller. The proper approach is to read the data into scalars and assign the values to the individual array elements as follows:

```
A = 0. & P = 0. & T = 0. & V = 0.
FOR I = 0, 99 DO BEGIN
   READF, 1, A, P, T, V
   ALT[I] = A & PRES[I] = P & TEMP[I] = T & VELO[I] = V
ENDFOR
```

————————————————————————————————————————————————————

# Using Unformatted Input/Output

Unformatted input/output involves the direct transfer of data between a file and memory without conversion to and from a character representation. Unformatted input/output is used when efficiency is important and portability is not an issue. It is faster and requires less space than formatted input/output. IDL provides three procedures for performing unformatted input/output:

## READU

Reads unformatted data from the specified file unit.

## WRITEU

Writes unformatted data to the specified file unit.

## ASSOC

Maps an array structure to a logical file unit, providing efficient and convenient direct access to data.

This section discusses READU and WRITEU, while ASSOC is discussed in "Associated Input/Output" on page 461. The READU and WRITEU procedures provide IDL's basic unformatted input/output capabilities. They have the form:

```
READU, Unit, Var_1, ..., Var_n
WRITEU, Unit, Var_1, ..., Var_n
```

where

*Unit* — The logical file unit with which the input/output operation will be performed.

*$Var_i$* — One or more IDL variables (or expressions in the case of output).

The WRITEU procedure writes the contents of its arguments directly to the file, and READU reads exactly the number of bytes required by the size of its arguments. Both cases directly transfer binary data with no interpretation or formatting.

## Unformatted Input/Output of String Variables

Strings are the only basic IDL data type that do not have a fixed size. A string variable has a dynamic length that is dependent only on the length of the string currently assigned to it. Thus, although it is always possible to know the length of the

other types, string variables are a special case. IDL uses the following rules to determine the number of characters to transfer:

## Input

Input enough bytes to fill the original length of the string. The length of the resulting string is truncated if the string contains a null byte.

## Output

Output the number of bytes contained in the string. This number is the same number returned by the STRLEN function and does not include a terminating null byte.

Note that these rules imply that when reading into a string variable from a file, you must know the length of the original string so as to be able to initialize the destination string to the correct length. For example, the following IDL statements produce the following output, because the receiving variable A was not long enough.

```
;Open a file.
OPENW, 1, 'temp.tmp'

;Write an 11-character string.
WRITEU, 1, 'Hello World'

;Rewind the file.
POINT_LUN, 1, 0

;Prepare a nine-character string.
A = '         '

;Read back in the string.
READU, 1, A

;Show what was input.
PRINT, A

CLOSE, 1
```

produce the following, because the receiving variable A was not long enough:

```
Hello Wor
```

The only solution to this problem is to know the length of the string being input. The following IDL statements demonstrate a useful "trick" for initializing strings to a known length:

```
;Open a file.
OPENW, 1, 'temp.tmp'
```

```
;Write an 11-character string.
WRITEU, 1, 'Hello World'

;Rewind the file.
POINT_LUN, 1, 0

;Create a string of the desired length initialized with blanks.
;REPLICATE creates a byte array of 11 elements, each element
;initialized to 32, which is the ASCII code for a blank. Passing
;this byte array to STRING converts it to a scalar string
;containing 11 blanks.
A = STRING(REPLICATE(32B,11))

;Read in the string.
READU, 1, A

;Show what was input.
PRINT, A

CLOSE, 1
```

This example takes advantage of the special way in which the BYTE and STRING functions convert between byte arrays and strings. See the description of the BYTE and STRING functions for additional details.

# Example: Reading C-Generated Unformatted Data with IDL

The following C program produces a file containing employee records. Each record stores the first name of each employee, the number of years he has been employed, and his salary history for the last 12 months.

```c
#include <stdio.h>

main()
{
    static struct rec {
            char name[32]; /* Employee's name */
            int years;     /* # of years with company */
            float salary[12]; /* Salary for last 12 months */
    } employees[] = {
{ {'B','u','l','l','w','i','n','k','l','e'}, 10,
    {1000.0, 9000.97, 1100.0, 0.0, 0.0, 2000.0,
  5000.0, 3000.0, 1000.12, 3500.0, 6000.0, 900.0} },{
{'B','o','r','r','i','s'}, 11,
    {400.0, 500.0, 1300.10, 350.0, 745.0, 3000.0,
  200.0, 100.0, 100.0, 50.0, 60.0, 0.25} },
```

```
        { {'N','a','t','a','s','h','a'}, 10,
             {950.0, 1050.0, 1350.0, 410.0, 797.0, 200.36,
          2600.0, 2000.0, 1500.0, 2000.0, 1000.0, 400.0} },
        { {'R','o','c','k','y'}, 11,
             {1000.0, 9000.0, 1100.0, 0.0, 0.0, 2000.37,
          5000.0, 3000.0, 1000.01, 3500.0, 6000.0, 900.12}}
        };

        FILE *outfile;

           outfile = fopen("data.dat", "w");
               (void) fwrite(employees, sizeof(employees), 1, outfile);
               (void) fclose(outfile);
        }
```

Running this program creates the file *data.dat* containing the employee records. The following IDL statements can be used to read and print this file:

```
;Create a string with 32 characters so that the proper number of
;characters will be input from the file. REPLICATE is used to
;create a byte array of 32 elements, each containing the ASCII code
;for a space (32). STRING turns this byte array into a string
;containing 32 blanks.
STR32 = STRING(REPLICATE(32B, 32))

;Create an array of four employee records to receive the input
;data.
A = REPLICATE({EMPLOYEES, NAME:STR32, YEARS:0L, $
   SALARY:FLTARR(12)}, 4)

;Open the file for input.
OPENR, 1, 'data.dat'

;Read the data.
READU, 1, A

CLOSE, 1

;Show the results.
PRINT, A
```

Executing these IDL statements produces the following output:

```
{ Bullwinkle          10
1000.00     9000.97     1100.00     0.00000     0.00000     2000.00
5000.00     3000.00     1000.12     3500.00     6000.00     900.000
}{Borris              11
400.000     500.000     1300.10     350.000     745.000     3000.00
200.000     100.000     100.000     50.0000     60.0000     0.250000
}{ Natasha            10
950.000     1050.00     1350.00     410.000     797.000     200.360
2600.00     2000.00     1500.00     2000.00     1000.00     400.000
}{ Rocky              11
1000.00     9000.00     1100.00     0.00000     0.00000     2000.37
5000.00     3000.00     1000.01     3500.00     6000.00     900.120
}
```

# Example: Reading IDL-Generated Unformatted Data with C

The following IDL program creates an unformatted data file containing a 5 x 5 array of floating-point values:

```
;Open a file for output.
OPENW, 1, 'data.dat'

;Write 5x5 array with each element set to its 1-dimensional index.
WRITEU, 1, FINDGEN(5, 5)

CLOSE, 1
```

This file can be read and printed by the following C program:

```c
#include <stdio.h>

main()
{
   float data[5][5];
      FILE *infile; int i, j;
      infile = fopen("data.dat", "r");
      (void) fread(data, sizeof(data), 1, infile);
      (void) fclose(infile);
      for (i = 0; i < 5; i++) {
         for (j = 0; j < 5; j++) {
            printf("%8.1f", data[i][j]);
            printf("\n");
         }
      }
}
```

Running this program gives the following output:

```
   0.0      1.0      2.0      3.0      4.0
   5.0      6.0      7.0      8.0      9.0
  10.0     11.0     12.0     13.0     14.0
  15.0     16.0     17.0     18.0     19.0
  20.0     21.0     22.0     23.0     24.0
```

# Example: Reading a Sun Rasterfile from IDL

Sun computers use rasterfiles to store scanned images. This example shows how to read such an image and display it using IDL. In the interest of keeping the example brief, a number of simplifications are made, no error checking is performed, and only 8-bit deep rasterfiles are handled. See the READ_SRF procedure (the file read_srf.pro in the lib subdirectory of the IDL distribution) for a complete example. The format used for rasterfiles is documented in the C header file /usr/include/rasterfile.h. That file provides the following information:

Each file starts with a fixed header that describes the image. In C, this header is defined as follows:

```
struct rasterfile{
     int ras_magic; /* magic number */
     int ras_width; /* width (pixels) of image */
     int ras_height; /* height (pixels) of image */
     int ras_depth; /* depth (1, 8, or 24 bits) */
     int ras_length; /* length (bytes) of image */
     int ras_type; /* type of file */
     int ras_maptype; /* type of colormap */
     int ras_maplength; /* length(bytes) of colormap */ };
```

The color map, if any, follows directly after the header information. The image data follows directly after the color map.

The following IDL statements read an 8-bit deep image from the file ras.dat:

```
;Define IDL structure that matches the Sun-defined rasterfile
;structure. A C int variable on a Sun corresponds to an IDL LONG
;int.
h = {rasterfile, magic:0L, width:0L, height:0L, depth: 0L,$
   length:0L, type:0L, maptype:0L, maplength:0L}

;Open the file, allocating a file unit at the same time.
OPENR, unit, file, /GET_LUN

;Read the header information.
READU, unit, h
```

```
   ;Is there a color map?
   IF ((h.maptype EQ 1) AND (h.maplength NE 0) ) THEN BEGIN

      ;Calculate length of each vector.
      maplen = h.maplength/3

      ;Create three byte vectors to hold the color map.
      r=(g=(b=BYTARR(maplen, /NOZERO)))

      ;Read the color map.
     READU, unit, r, g, b

   ENDIF

   ;Create a byte array to hold image.
   image = BYTARR(h.width, h.height, /NOZERO)

   ;Read the image.
   READU, unit, image

   ;Free the previously-allocated Logical Unit Number and close the
   ;file.
   FREE_LUN, unit
```

# Portable Unformatted Input/Output

Normally, unformatted input/output is not portable between different machine architectures because of differences in the way various machines represent binary data. However, it is possible to produce binary files that are portable by specifying the XDR keyword with the OPEN procedures. XDR (for eXternal Data Representation) is a scheme under which all binary data is written using a standard "canonical" representation. All machines supporting XDR understand this standard representation and have the ability to convert between it and their own internal representation.

XDR represents a compromise between the extremes of unformatted and formatted input/output:

- It is not as efficient as purely unformatted input/output because it does involve the overhead of converting between the external and internal binary representations.

- It is still much more efficient than formatted input/output because conversion to and from ASCII characters is much more involved than converting between binary representations.

- It is much more portable than purely unformatted data, although it is still limited to those machines that support XDR. However, XDR is freely available and can be moved to any system.

## XDR Considerations

The primary differences in the way IDL input/output procedures work with XDR files, as opposed to files opened normally are as follows:

- To use XDR, you must specify the XDR keyword when opening the file.

- The only input/output data transfer routines that can be used with a file opened for XDR are READU and WRITEU.

- XDR converts between the internal and standard external binary representations for data instead of simply using the machine's internal representation.

- Since XDR adds extra "bookkeeping" information to data stored in the file and because the binary representation used may not agree with that of the machine being used, it does not make sense to access an XDR file without using XDR.

- OPENW and OPENU normally open files for both input and output. However, XDR files can only be opened in one direction at a time. Thus, using these procedures with the XDR keyword results in a file open for output only. OPENR works in the usual way.

- The length of strings is saved and restored along with the string. This means that you do not have to initialize a string of the correct length before reading a string from the XDR file. (This is necessary with normal unformatted input/output and is described in "Using Unformatted Input/Output" on page 449).

- For efficiency reasons, byte arrays are transferred as a single unit; therefore, byte variables must be initialized to the correct number of elements for the data to be input, or an error will occur. For example, given the statements,

```
;Open a file for XDR output.
OPENW, /XDR, 1, 'data.dat'

;Write a 10-element byte array.
WRITEU, 1, BINDGEN(10)

;Close the file and re-open it for input.
CLOSE, 1 & OPENR, /XDR, 1, 'data.dat'
```

then the statement,

```
;Try to read the first byte only.
B = 0B & READU, 1, B
```

results in the following error:

```
% READU: Error encountered reading from file unit: 1.
```

Instead, it is necessary to read the entire byte array back in one operation using a statement such as:

```
;Read the whole array back at once.
B=BYTARR(10) & READU, 1, B
```

This restriction does not exist for other data types.

## IDL XDR Conventions for Programmers

IDL uses certain conventions for reading and writing XDR files. If your only use of XDR is through IDL, you do not need to be concerned about these conventions because IDL takes care of it for you. However, programmers who want to create IDL-

compatible XDR files from other languages need to know the actual XDR routines
used by IDL for various data types. The following table summarizes this information.

| Data Type | XDR routine |
|---|---|
| Byte | xdr_bytes() |
| Integer | xdr_short() |
| Long | xdr_long() |
| Float | xdr_float() |
| Double | xdr_double() |
| Complex | xdr_complex() |
| String | xdr_counted_string() |
| Double Complex | xdr_dcomplex() |
| Unsigned Integer | xdr_u_short() |
| Unsigned Long | xdr_u_long() |
| 64-bit Integer | xdr_long_long_t() |
| Unsigned 64-bit Integer | xdr_u_long_long_t() |

*Table 18-13: XDR Routines Used by IDL*

The routines used for type COMPLEX, DCOMPLEX, and STRING are not primitive
XDR routines. Their definitions are as follows:

```
bool_t xdr_complex(xdrs, p)
     XDR *xdrs;
     struct complex { float r, i} *p;
{
  return(xdr_float(xdrs, (char *) &p->r) &&
        xdr_float(xdrs, (char *) &p->i));
}
bool_t xdr_dcomplex(xdrs, p)
     XDR *xdrs;
     struct dcomplex { double r, i} *p;
{
  return(xdr_double(xdrs, (char *) &p->r) &&
        xdr_double(xdrs, (char *) &p->i));
}
bool_t xdr_counted_string(xdrs, p)
     XDR *xdrs;
```

```
        char **p;
{
  int input = (xdrs->x_op == XDR_DECODE);
  short length;

  /* If writing, obtain the length */
  if (!input) length = strlen(*p);

  /* Transfer the string length */
  if (!xdr_short(xdrs, (char *) &length)) return(FALSE);

  /* If reading, obtain room for the string */
  if (input)
  {
      *p = malloc((unsigned) (length + 1));
         *p[length] = '\0'; /* Null termination */
  }
  /* If the string length is nonzero, transfer it */
  return(length ? xdr_string(xdrs, p, length) : TRUE);
}
```

# Example: Reading C-Generated XDR Data with IDL

The following C program produces a file containing different types of data using
XDR. The usual error checking is omitted for the sake of brevity.

```
#include <stdio.h>
#include <rpc/rpc.h>
[ xdr_complex() and xdr_counted_string() included here ]

main()
{
   static struct {        /* Output data */
        unsigned char c;
        short s;
        long l;
        float f;
        double d;
        struct complex { float r, i } cmp;
        char *str;
    }
    data = {1, 2, 3, 4, 5.0, { 6.0, 7.0}, "Hello" };
    u_int c_len = sizeof(unsigned char); /* Length of a char */
    char *c_data = (char *) &data.c;      /* Addr of byte field */
    FILE *outfile;                        /* stdio stream ptr */
    XDR xdrs;                             /* XDR handle */

    /* Open stdio stream and XDR handle */
```

```
    outfile = fopen("data.dat", "w");
    xdrstdio_create(&xdrs, outfile, XDR_ENCODE);

/* Output the data */
    (void) xdr_bytes(&xdrs, &c_data, &c_len, c_len);
    (void) xdr_short(&xdrs, (char *) &data.s);
    (void) xdr_long(&xdrs, (char *) &data.l);
    (void) xdr_float(&xdrs, (char *) &data.f);
    (void) xdr_double(&xdrs, (char *) &data.d);
    (void) xdr_complex(&xdrs, (char *) &data.cmp);
    (void) xdr_counted_string(&xdrs, &data.str);

/* Close XDR handle and stdio stream */
    xdr_destroy(&xdrs);
    (void) fclose(outfile);
}
```

Running this program creates the file *data.dat* containing the XDR data. The following IDL statements can be used to read this file and print its contents:

```
;Create structure containing correct types.
DATA={S, C:0B, S:0, L:0L, F:0.0, D:0.0D, CMP:COMPLEX(0), STR:''}

;Open the file for input.
OPENR, /XDR, 1, 'data.dat'

;Read the data.
READU, 1, DATA

;Close the file.
CLOSE, 1

;Show the results.
PRINT, DATA
```

Executing these IDL statements produces the output:

```
{  1        2            3        4.00000        5.0000000
(      6.00000,      7.00000) Hello}
```

For further details about XDR, consult the XDR documentation for your machine. Sun users should consult their *Network Programming* manual.

# Associated Input/Output

Unformatted data stored in files often consists of a repetitive series of arrays or structures. A common example is a series of images. IDL-associated file variables offer a convenient and efficient way to access such data.

An associated variable is a variable that maps the structure of an IDL array or structure variable onto the contents of a file. The file is treated as an array of these repeating units of data. The first array or structure in the file has an index of zero, the second has index one, and so on. Such variables do not keep data in memory like a normal variable. Instead, when an associated variable is subscripted with the index of the desired array or structure within the file, IDL performs the input/output operation required to access the data.

When their use is appropriate (the file consists of a sequence of identical arrays or structures), associated file variables offer the following advantages over READU and WRITEU for unformatted input/output:

- Input/output occurs when an associated file variable is subscripted. Thus, it is possible to perform input/output within an expression without a separate input/output statement.

- The size of the data set is limited primarily by the maximum possible size of the file containing the data instead of the maximum memory available. Data sets too large for memory can be accessed.

- There is no need to declare the maximum number of arrays or structures contained in the file.

- Associated variables offer transparent access to data. Direct access to any element in the file is rapid and simple—there is no need to calculate offsets into the file and/or position the file pointer prior to performing the input/output operation.

An associated file variable is created by assigning the result of the ASSOC function to a variable. See "ASSOC" (*IDL Reference Guide*) for details.

## Example of Using Associated Input/Output

Assume that a file named *data.dat* exists, and that this file contains a series of 10 x 20 arrays of floating-point data. The following two IDL statements open the file and create an associated file variable mapped to the file:

```
;Open the file.
OPENU, 1, 'data.dat'
```

```
;Make a file variable. Using the NOZERO keyword with FLTARR
;increases efficiency.
A = ASSOC(1, FLTARR(10, 20, /NOZERO))
```

The order of these two statements is not important—it would be equally valid to call ASSOC first, and then open the file. This is because the association is between the variable and the logical file unit, not the file itself. It is also legitimate to close the file, open a new file using the same LUN, and then use the associated variable without first executing a new ASSOC. Naturally, an error occurs if the file is not open when the file variable is subscripted in an expression or if the file is open for the wrong type of access (for example, trying to assign to an associated file variable linked with a file opened for read-only access).

As a result of executing the two statements above, the variable A is now an associated file variable. Executing the statement,

```
HELP, A
```

gives the following response:

```
A               FLOAT     = File<data.dat> Array(10, 20)
```

The associated variable A maps the structure of a 10 x 20, floating-point array onto the contents of the file *data.dat*. Thus, the response from the HELP procedure shows it as having the structure of a two-dimensional array. An associated file variable only performs input/output to the file when it is subscripted. Thus, the following two IDL statements do not cause input/output to happen:

```
B = A
```

This assignment does not transfer data from the file to variable B because A is not subscripted. Instead, B becomes an associated file variable with the same structure, and to the same logical file unit, as A.

```
B = 23
```

This assignment does not result in the value 23 being transferred to the file because variable B (which became a file variable in the previous statement) is not subscripted. Instead, B becomes a scalar integer variable containing the value 23. It is no longer an associated file variable.

# Reading Data from Associated Files

Once a variable has been associated with a file, data are read from the file whenever the associated variable appears in an expression with a subscript. The position of the array or structure read from the file is given by the value of the subscript. The

following IDL statements assume that the associated file variable A is defined as in the previous section, and give some examples of using file variables:

```
;Copy the contents of the first array into normal variable Z. Z is
;now a 10 x 20, floating-point array.
Z = A[0]

;Form the sum of the first 10 arrays. (Z was initialized in the
;previous statement to the value of the first array. This statement
;adds the following nine to it.) Note the use of the compound
;operator += to avoid creating a new copy of Z each time we add a
;new array.
FOR I = 1, 9 DO Z += A[I]

;Read fourth array and plot it.
PLOT, A[3]

;Subtract array four from array five, and plot the result. The
;result of the subtraction is then discarded.
PLOT, A[5] - A[4]
```

# Writing Data to Associated Files

When a subscripted associated variable appears on the left side of an assignment statement, the expression on the right side is written into the file at the given array position:

```
;Sets sixth record to zero.
A[5] = FLTARR(10, 20)

;Write ARR into sixth record after any necessary type conversions.
A[5] = ARR

;Averages records J and J+1, and writes the result into record J.
A[J] = (A[J] + A[J + 1])/2
```

# Multiple Subscripts With Associated File Variables

Usually, when subscripts are used with associated file variables, only a single subscript is present, specifying an array within the associated file. This is the most efficient way to access associated file variables. However, IDL allows you to specify individual elements within the selected array using multiple subscripts. When multiple subscripts are present with an associated file variable, the rightmost subscript selects the array within the file, and the other subscripts specify the specific element within that array.

For example, consider the following statement using the variable A defined above:

```
Z = A[0,0,1]
```

This statement assigns the value of element [0,0] of the second array within the file to the variable Z. The rightmost subscript is interpreted as the subscript of the array within the file, causing IDL to read the entire array into memory. This resulting array expression is then further subscripted by the remaining subscripts.

Similarly, the statement:

```
A[2,3,4] = 45
```

assigns the value 45 to element [2,3] of the fifth array within the file. When a file variable is referenced, the last (and possibly only) subscript denoting the element within that array must be a simple subscript. Other subscripts and subscript ranges, except the last, have the same meaning as when used with normal array variables.

An implicit extraction of an element or subarray in a data record can also be performed. For example:

```
; Variable A associates the file open on unit 1 with the records of
;200-element, floating-point vectors.
A = ASSOC(1, FLTARR(200))

; Then, X is set to the first 100 points of record number 2, the
; third record of the file.
X = A[0:99, 2]

; Set the 24th point of record 16 to 12.
A[23, 16] = 12

; Increment points 10 to 199 of record 12. Points 0 to 9 of the
; record remain unchanged.
A[10, 12] = A[10:*, 12]+1
```

**Note**  ────────────────────────────────────────────────────────────────

Although the ability to directly refer to array elements within an associated file can be convenient, it can also be very slow because every access to an array element causes the entire array to be transferred to or from memory. Unless only one operation on the array is required, it is faster to assign the contents of the array to a normal variable by subscripting the file variable with a single subscript, and then access the individual array elements within the normal variable as needed. If you make changes to the value of the normal variable that should be reflected in the file, a final assignment to the associated variable, indexed with a single subscript, can be used to update the file and complete the operation.

─────────────────────────────────────────────────────────────────────────

# Files with Multiple Structures

The same file may be associated with a number of different structures. Assume a number of 128 x 128-byte images are contained on a file. The statement,

```
ROW = ASSOC(1, BYTARR(128))
```

will map the file into rows of 128 bytes each. ROW[3] is the fourth row of the first image, while ROW[128] is the first row of the second image. The statement,

```
IMAGE = ASSOC(1, BYTARR(128, 128))
```

maps the file into entire images; IMAGE[4] will be the fifth image.

# Offset Parameter

The *Offset* parameter to ASSOC specifies the position in the file at which the first array starts. This parameter is useful when a file contains a header followed by data records. For example, if a file uses the first 1,024 bytes of the file to contain header information, followed by 512 x 512-byte images, the statement,

```
IMAGE = ASSOC(1, BYTARR(512, 512), 1024)
```

sets the variable IMAGE to access the images while skipping the header.

# Efficiency

Arrays are accessed most efficiently if their length is an integer multiple of the block size of the filesystem holding the file. Common values are powers of 2, such as 512, 2K (2048), 4K (4096), or 8K (8192) bytes. For example, on a disk with 512-byte blocks, one benchmark program required approximately one-eighth of the time required to read a 512 x 512-byte image that started and ended on a block boundary, as compared to a similar program that read an image that was not stored on even block boundaries.

Each time a subscripted associated variable is referenced, one or more records are read from or written to the file. Therefore, if a record is to be accessed more than a few times, it is more efficient to read the entire record into a variable. After making the required changes to the in-memory variable, it can be written back to the file if necessary.

# Unformatted Data from UNIX FORTRAN Programs

Unformatted data files generated by FORTRAN programs under UNIX contain an extra long word before and after each logical record in the file. ASSOC does not interpret these extra bytes but considers them to be part of the data. This is true even if the F77_UNFORMATTED keyword is specified on the OPEN statement. Therefore, ASSOC should not be used with such files. Instead, such files should be processed using READU and WRITEU. An example of using IDL to read such data is given in "Using Unformatted Input/Output" on page 449.

# File Manipulation Operations

IDL provides a variety of routines that allow you to retrieve information about and manipulate files and directories. See the following topics:

- Chapter 3, "Importing and Writing Data into Variables" (*Using IDL*) describes various methods of accessing files

- "General File Access" (*IDL Quick Reference*) provides a complete list of routines that allow you to access, locate, modify, and get information about files

## Working with UNIX Links

On UNIX platforms, you can create file links, both regular (hard) and symbolic. A hard link is a directory entry that references a file. UNIX allows multiple such links to exist simultaneously, meaning that a given file can be referenced by multiple names. The following limitations on hard links are enforced by the operating system:

- Hard links may not span file systems, as hard linking is only possible within a single file system.

- Hard links may not be created between directories, as doing so has the potential to create infinite circular loops within the hierarchical Unix file system. Such loops will confuse many system utilities, and can even cause file system damage.

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

Use the FILE_LINK procedure to create hard and soft links on UNIX systems. See "FILE_LINK" (*IDL Reference Guide*) for details.

Use the FILE_READLINK procedure to retrieve the path to a file referenced by a UNIX symbolic link. See "FILE_READLINK" (*IDL Reference Guide*) for details.

Use the FILE_SAME function to determine whether two file names refer to the same underlying file. See "FILE_SAME" (*IDL Reference Guide*) for details.

# Reading and Writing FORTRAN Data

The standard FORTRAN *unformatted sequential* file input/output mechanism performs file input and output by reading and writing blocks of data from (or to) a file as *logical records*. To read data, the FORTRAN program asks for the next logical record from an open file; the operating system is then responsible for determining how much data should be retrieved from the file. This system works well for operating systems like VMS, which organize files into records and can thus keep track of where logical blocks of data begin and end.

In contrast, the UNIX and Microsoft Windows operating systems supported by IDL treat files as an uninterrupted stream of bytes. In order to reconcile the FORTRAN need for logical records with these *stream files*, FORTRAN compilers for UNIX and Microsoft Windows provide a mechanism to add a longword integer count of the number of bytes in each logical record. This mechanism allows FORTRAN-generated data files that treat data as a series of logical records to be read on platforms that use stream files.

The F77_UNFORMATTED keyword to the OPEN procedures informs IDL that the file contains unformatted data demarcated by logical record identifiers. When a file is opened with this keyword, IDL interprets the longword counts properly and is able to read the logical records. Similarly, IDL can write data using the logical record format using the F77_UNFORMATTED keyword.

Use the F77_UNFORMATTED keyword if your IDL program is reading data that contain embedded longword logical record separators, or if your program is writing data that will be read by a FORTRAN program that reads unformatted sequential files.

**Note** ───────────────────────────────────────────────────
On 64-bit machines, some Fortran compilers will insert record markers that are 64-bit integers instead of the standard 32-bit integers. When reading FORTRAN data, IDL will attempt to recognize the presence of 64-bit record markers and switch to the appropriate format. When writing unformatted Fortran files, IDL will continue to use 32-bit record markers.
───────────────────────────────────────────────────────────

**Note** ───────────────────────────────────────────────────
Direct-access FORTRAN I/O does not write data using logical records, but simply transfers binary data to or from the file.
───────────────────────────────────────────────────────────

# **Reading Data from a FORTRAN File**

The following FORTRAN program, when run on a UNIX or Microsoft Windows system (that is, an operating system that uses stream files), produces a file containing a five-column by three-row array of floating-point values with each element set to its one-dimensional subscript:

```
     PROGRAM ftn2idl

     INTEGER i, j
     REAL data(5, 3)

     OPEN(1, FILE="ftn2idl.dat", FORM="unformatted")
       DO 100 j = 1, 3
         DO 100 i = 1, 5
           data(i,j) = ((j - 1) * 5) + (i - 1)
           print *, data(i,j)
100 CONTINUE
     WRITE(1) data
     END
```

Running this program creates the file *ftn2idl.dat* containing the unformatted array. The following IDL statements can be used to read this file and print out its contents:

```
;Create an array to contain the fortran array.
data = FLTARR(5,3)

;Open the fortran-generated file. The F77_UNFORMATTED keyword is
;necessary so that IDL will know that the file contains unformatted
;data produced by a UNIX FORTRAN program.
OPENR, lun, 'ftn2idl.dat', /GET_LUN, /F77_UNFORMATTED

;Read the data in a single input operation.
READU, lun, data

;Release the logical unit number and close the fortran file.
FREE_LUN, lun

;Print the result.
PRINT, data
```

Executing these IDL statements produces the following output:

```
0.00000      1.00000      2.00000      3.00000      4.00000
5.00000      6.00000      7.00000      8.00000      9.00000
10.0000      11.0000      12.0000      13.0000      14.0000
```

Because unformatted data produced by FORTRAN unformatted WRITE statements on an operating system that uses stream files are interspersed with extra information

before and after each logical record, it is important that the IDL program read the data in the same way that the FORTRAN program wrote it. For example, consider the following attempt to read the above data file one row at a time:

```
;Create an array to contain one row of the FORTRAN array.
data = FLTARR(5, /NOZERO)

OPENR, lun, 'ftn2idl.dat', /GET_LUN, /F77_UNFORMATTED

;One row at a time.
FOR I = 0, 4 DO BEGIN

   ;Read a row of data.
   READU, lun, data

   ;Print the row.
   PRINT, data
ENDFOR

;Close the file.
FREE_LUN, lun
```

Executing these IDL statements produces the output:

```
0.00000      1.00000      2.00000      3.00000      4.00000
% READU: End of file encountered. Unit: 100
         File: ftn2idl.dat6
% Execution halted at $MAIN$(0).
```

Here, IDL attempted to read the single logical record written by the FORTRAN program as if it were written in five separate records. IDL hit the end of the file after reading the first five values of the first record.

# Writing Data to a FORTRAN File

The following IDL statements create a five-column by three-row array of floating-point values with each element set to its one-dimensional subscript, and writes the array to a data file suitable for reading by a FORTRAN program:

```
;Create the array.
data = FINDGEN(5,3)

;Open a file for writing. Note that the F77_UNFORMATTED keyword is
;necessary to tell IDL to write the data in a format readable by a
;FORTRAN program.
OPENW, lun, 'idl2ftn.dat', /GET_LUN, /F77_UNFORMATTED

;Write the data.
WRITEU, lun, data
```

```
          ;Close the file.
          FREE_LUN, lun
```

The following FORTRAN program reads the data file created by IDL:

```
      PROGRAM idl2ftn

      INTEGER i, j
      REAL data(5, 3)

      OPEN(1, FILE="idl2ftn.dat", FORM="unformatted")
      READ(1) data
        DO 100 j = 1, 3
          DO 100 i = 1, 5
            PRINT *, data(i,j)
  100 CONTINUE
      END
```

# Platform-Specific File I/O Information

Special considerations for file access on UNIX and Windows platforms are covered in the following sections:

## UNIX-Specific Information

Under UNIX, a file is read or written as an uninterrupted stream of bytes — there is no record structure at the operating system level. (By convention, records of text are simply terminated by the linefeed character, which is referred to as "newline.") It is possible to move the current file pointer to any arbitrary position in the file and to begin reading or writing data at that point. This simplicity and generality form a system in which any type of file can be manipulated easily using a small set of file operations.

## Windows-Specific Information

Under Microsoft Windows, a file is read or written as an uninterrupted stream of bytes — there is no record structure at the operating system level. Lines in a Windows text file are terminated by the character sequence CR LF (carriage return, line feed).

The Microsoft C runtime library considers a file to be in either binary or text mode, and its behavior differs depending on the current mode of the file. The programmer confusion caused by this distinction is a cause of many C/C++ program bugs. Programmers familiar with this situation may be concerned about how IDL handles read and write operations. IDL is not affected by this quirk of the C runtime library, and no special action is required to work around it. Read/write operations are handled the same in Windows as in Unix: when IDL performs a formatted I/O operation, it reads/writes the CR/LF line termination. When it performs a binary operation, it simply reads/writes raw data.

Versions of IDL prior to IDL 5.4 (5.3 and earlier), however, *were* affected by the text/binary distinction made by the C library. The BINARY and NOAUTOMODE keywords to the OPEN procedures were provided to allow the user to change IDL's default behavior during read/write operations. In IDL 5.4 and later versions, these keywords are no longer necessary. They continue to be accepted in order to allow older code to compile and run without modification, but they are completely ignored and can be safely removed from code that does not need to run on those older versions of IDL.

# Chapter 19
# Using Language Catalogs

The following topics are covered in this chapter:

# What Is a Language Catalog?

A *language catalog* is a set of text strings in a particular language, created as key name/value pairs. Applications can use these catalogs to fill in the names of menu items, buttons, and other elements of a user interface, for example. The use of different language catalogs, then, can support an application's internationalization: for example, letting a user decide what language to use for installing and running the application.

There are two main advantages of using a language catalog in a separate file, rather than having these strings embedded in the application:

- The strings do not consume memory until the application loads them

- You can edit the catalog to add new languages and new strings without directly involving the application

In addition, because the language catalogs are in XML format, you can easily read and edit the files in any text editor.

Implementing language catalog functionality requires two parts:

- The creation of a language catalog (`.cat`) file, which contains the name/value pairs in the desired languages. See "Creating a Language Catalog File" on page 475 for information on requirements for a language catalog file's structure.

- The creation of an IDLffLangCat object, which provides access and use of the keys in the catalog file. See "Using the IDLffLangCat Class" on page 478 for information on creating and using a language catalog object.

# Creating a Language Catalog File

A language catalog (`.cat`) file contains the XML that defines the text strings as key name/value pairs within a single `<IDLffLangCat>` tag. The tag can contain four optional attributes, as described in Table 19-1.

| Attribute | Description |
|---|---|
| APPLICATION | Name of the application that will use the keys in the file |
| VERSION | Version of IDLffLangCat for which the file was created |
| DATE | Date of the file's creation or last modification, as desired |
| AUTHOR | Author of the file |

*Table 19-1: IDLffLangCat Tag Attributes*

**Note** ────────────────────────────────────────

You cannot perform queries on VERSION, DATE, and AUTHOR. These attributes are more like XML comments on the tag; they are informational only.

────────────────────────────────────────

**Note** ────────────────────────────────────────

For more information on XML, see "About XML" on page 486.

────────────────────────────────────────

The following XML snippet, extracted from the iTools menu catalog file that comes with the IDL installation, illustrates the basic file structure:

```
<IDLffLangCat APPLICATION="itools menu" VERSION="1.0"
   AUTHOR="ITT">
   <LANGUAGE NAME="English">
      <KEY NAME="Menu:File">File</KEY>
      <KEY NAME="Menu:File:New">New</KEY>
      <KEY NAME="Menu:File:Open">Open...</KEY>
   </LANGUAGE>
</IDLffLangCat>
```

The `<IDLffLangCat>` tag can contain any number of LANGUAGE tags. Each LANGUAGE tag must have a NAME attribute denoting the language contained therein.

Each LANGUAGE tag can contain any number of KEY tags. Each KEY tag must have a NAME attribute denoting the name of the key.

**Note** —————————————————————————————————————————————
All text between the open and close KEY tags will be part of the string returned by
the query, including any line feeds, carriage returns, and spaces.
———————————————————————————————————————————————————

The catalog file can contain keys for one or more languages. Whether there is a single
catalog file containing multiple languages, or multiple catalog files, each containing a
single language, is personal preference.

By keeping each language separate in the tag definition, you can easily cut and paste
an entire block and then change the strings of one language to another language while
keeping all the keys intact. This technique also allows for the possibility of having
different languages in separate files. Note that the keys in any one language need not
match those of another language (although in most cases they will).

**Note** —————————————————————————————————————————————
IDL supports catalog files written in 8-bit strings (which can be encoded for
languages using special marks; see below). Also, you must have the corresponding
fonts installed on your machine before you can use a particular language.
———————————————————————————————————————————————————

**Note** —————————————————————————————————————————————
If your language has accent marks such as those in French, you might need to
modify the catalog file to support those encodings. In general, you should use the
encoding appropriate for your catalog's language. For more information, see
"IDLffLangCat" (*IDL Reference Guide*).
———————————————————————————————————————————————————

# Storing and Loading Language Catalog Files

The catalog files included with IDL are in the /resource/langcat directory of the
IDL installation and end in a .cat extension. These files contain the English keys for
iTools menus, dialogs, and messages and are provided to support the use of
applications using iTools functionality in other languages. All catalog files must end
with the .cat extension if APP_NAME is used to locate the files.

You can create custom catalog files and place them in a location of your choice. You
typically use a full path to access these catalog files through the creation of an
IDLffLangCat object (see "Using the IDLffLangCat Class" on page 478 for more
information).

You can specify a catalog either by giving the full path of the catalog file or files, or
by providing an application name or names and, optionally, an application path or
paths. If no path is specified, only the current directory is searched. For all

application paths, all `.cat` files found in any of the directories listed are searched for all given applications.

On a similar note, if IDL finds a duplicate key name while loading keys, IDL will use the string corresponding to the last key found with the given name.

# Using the IDLffLangCat Class

You use the IDLffLangCat class to find and load an XML language catalog. The class also provides methods for retrieving text strings by matching key names.

## Creating a Language Catalog Object

The IDL installation comes with an English language catalog for the iTools menu, called `itoolsmenu_eng.cat`, in the `/resource/langcat` directory of the IDL installation. To load the keys in this file:

```
oLangCat = OBJ_NEW( 'IDLffLangCat', 'ENGLISH', $
   APP_NAME='itools menu', $
   APP_PATH=FILEPATH('', $
      SUBDIRECTORY=['resource','langcat','itools']), /VERBOSE )
```

This command searches the given directory for language catalog keys in English that match the application of 'itools menu.' In fact, if there are any other language catalog files, besides `itoolsmenu_eng.cat`, containing keys whose LANGUAGE value is 'ENGLISH' and APPLICATION value is 'itools menu,' the object adds those keys as well. The matches must be exact in that 'itools menu2,' for example, is not a match; however, the matching is not case-sensitive (i.e., 'ENGLISH' and 'English' are both matches for LANGUAGE).

**Note**
Whenever the object encounters a key (or language) that already exists, the key (or language) is overwritten with the new value.

The VERBOSE flag on the command sends all catalog-loading messages to the IDL Workbench output window. This list contains details resulting from the object's initialization (the names and numbers of keys loaded, keys overwritten, etc.).

## Adding Application Keys

You might want to add keys for a different application to an existing language catalog object. To do so:

```
retval = oLangCat->AppendCatalog( APP_NAME='itools ui', $
   APP_PATH=FILEPATH('', $
   SUBDIRECTORY=['resource','langcat','itools']) )
```

This command searches the given directory for keys matching an APPLICATION value of 'itools ui' and appends them to `oLangCat`. The method returns a value indicating success or failure of the operation.

# Getting and Setting Languages

To return the available languages in a language catalog object:

```
oLangCat->GetProperty, AVAILABLE_LANGUAGES=availLangs
```

This command stores the list of available languages as a string array in `availLangs`.

To set the current language of a language catalog object (the language used for query searches and matching):

```
oLangCat->SetProperty, LANGUAGE='English'
```

You can use these two methods for getting and setting other properties of a language catalog object. For the list of available object properties, see "IDLffLangCat Properties" in the *IDL Reference Guide* manual.

Comparisons such as those done with the Query method (see "Performing Queries" on page 479) are case insensitive, but the values returned by the GetProperty method are exactly as the last encountered value. The exception is that all key names are returned in uppercase. For example, if File 1 has `LANGUAGE='English'` and File 2 has `LANGUAGE='engLISh'`, then `'engLISh'` will be returned, although only one `ENGLISH` language exists in the current catalog.

# Performing Queries

To populate the text fields of a widget or other interface object, for example, you can query a language catalog object for key values it contains. IDL performs the search on the `NAME` attribute of the keys; matches are not case-sensitive.

```
keyVal = oLangCat->Query( 'Menu:File:New', $
    DEFAULT_STRING='Key not found' )
```

This command searches `oLangCat` for keys with the `NAME` value of 'Menu:File:New' and returns the match in `keyVal`. If `oLangCat` finds a match in the current language, `keyVal` will hold that value string. If a given key does not exist in the current language, the default language is queried (if one exists). If there are still no matches, the default string is returned.

You can use more than one key in a query by passing an array of strings to the Query method (e.g., `['Menu:File:New','Menu:File:Open']`). Similarly, you can supply an array of strings for the `DEFAULT_STRING` keyword. In such a case, only those values in the array whose indices match the missing keys will be returned. If you do not specify `DEFAULT_STRING`, a null string will be returned instead.

# Destroying a Language Catalog Object

You can destroy a catalog object as you would any other IDL object, as follows:

```
OBJ_DESTROY, oLangCat
```

Destroying a language catalog object does not affect any files from which the object drew its keys.

# Widget Example

This example creates a widget with two buttons whose text strings change between two languages, depending on the selection from a drop-down list.

The following language catalogs are two separate files (as denoted by the <IDLffLangCat> tag for each) and should be placed on your system as such.

```
<?xml version="1.0"?>
<!-- $Id: myButtonsText.eng.cat,v 1.1 2004 rsiDoc Exp $ -->
<IDLffLangCat APPLICATION="myOpenButtons" VERSION="1.0"
   AUTHOR="ITT">
   <LANGUAGE NAME="English">
      <KEY NAME="Button:OpenFile">Open File</KEY>
      <KEY NAME="Button:OpenFolder">Open Folder</KEY>
   </LANGUAGE>
</IDLffLangCat>

<?xml version="1.0"?>
<!-- $Id: myButtonsText.fr.cat,v 1.1 2004 rsiDoc Exp $ -->
<IDLffLangCat APPLICATION="myOpenButtons" VERSION="1.0"
   AUTHOR="ITT">
   <LANGUAGE NAME="French">
      <KEY NAME="Button:OpenFile">Ouvrir le Fichier</KEY>
      <KEY NAME="Button:OpenFolder">Ouvrir le Dossier</KEY>
   </LANGUAGE>
</IDLffLangCat>
```

To use the following code, save it in a .pro file. You do not have to run it from the same directory containing the language catalog files.

```
; Routine to change the language of the button labels.
PRO button_language_change, pstate
   vLangString = (*pstate).vlang

   ; Access the language catalog to retrieve string values.
   oLangCat = OBJ_NEW( 'IDLffLangCat', vLangString, $
      APP_NAME='myOpenButtons' , APP_PATH=(*pstate).vpath)
   ; Access and store language-specific strings in the structure.
   strOpenFile = oLangCat->Query( 'Button:OpenFile' )
   strOpenFolder = oLangCat->Query( 'Button:OpenFolder' )
   WIDGET_CONTROL, (*pstate).pb1, SET_VALUE=strOpenFile
   WIDGET_CONTROL, (*pstate).pb2, SET_VALUE=strOpenFolder
END

; Event handler for 'Open File' button.
PRO button_file, event
   sFile = DIALOG_PICKFILE( TITLE='Select image file' )
```

```
         END

         ; Event handler for 'Open Folder' button.
         PRO button_folder, event
            sFolder = DIALOG_PICKFILE( /DIRECTORY, $
               TITLE='Choose the directory in which to store the data' )
         END

         ; Event handler for 'Language' droplist.
         PRO button_language_event, event
            WIDGET_CONTROL, event.top, GET_UVALUE = pstate
            ; Access user's language selection and store it in the pointer.
            IF event.index EQ 0 THEN (*pstate).vlang = 'English'
            IF event.index EQ 1 THEN (*pstate).vlang = 'French'
            ; Call the procedure to change the button text.
            button_language_change, pstate
         END

         ; Widget-creation procedure
         PRO button_language
            ; Prompt for path to catalog files
            vpath=dialog_pickfile( TITLE='Select directory that ' + $
               'contains *.cat files', /DIRECTORY )
            IF vpath EQ '' THEN return

            ; Create a top level base. Not specifying tab mode uses default
            ; value of zero (do not allow widgets to receive or lose focus).
            tlb = WIDGET_BASE( /COLUMN, TITLE = "Language Change", $
               XSIZE=220, /BASE_ALIGN_CENTER )
            ; Create the button widgets.
            bbase = WIDGET_BASE( tlb, /COLUMN )
            pb1 = WIDGET_BUTTON( bbase, VALUE='Open File', $
               UVALUE='openFile', XSIZE=105, EVENT_PRO='button_file' )
            pb2 = WIDGET_BUTTON( bbase, VALUE='Open Folder', $
               UVALUE='openFolder', XSIZE=105, EVENT_PRO='button_folder' )
            ; Create a drop-down list indicating available catalogs.
            vLangList = ['English', 'French']
            langDrop = WIDGET_DROPLIST( tlb, VALUE=vLangList, $
               TITLE='Language' )
            ; Draw the widgets and activate events.
            WIDGET_CONTROL, tlb, /REALIZE

            ; Create the state structure.
            state = { $
               pb1:pb1, $
               pb2:pb2, $
               vlang:'', $
               vpath:vpath $
            }
```

```
pstate = PTR_NEW( state, /NO_COPY )
WIDGET_CONTROL, tlb, SET_UVALUE=pstate
XMANAGER, 'button_language', tlb

; Clean up pointers.
PTR_FREE, pstate
END
```

# Chapter 20
# Using the XML Parser Object Class

The following topics are covered in this chapter:

# About XML

XML (eXtensible Markup Language) provides a set of rules for defining semantic tags that can describe virtually any type of data in a text file. Data stored in XML-format files is both human- and machine-readable, and is often relatively easy to interpret either visually or programmatically. The structure of data stored in an XML file is described by either a Document Type Definition (DTD) or an XML schema, which can either be included in the file itself or referenced from an external network location.

The IDL parsers support the following encodings: UTF-8, USASCII, ISO8859-1, UTF-16, UTF-16BE, UTF-16LE, UCS-4, UCS-4BE, UCS-4LE, WINDOWS-1252, IBM1140, IBM037, and IBM1047.

**Note** ──────────────────────────────────────────────────────

IDL can parse XML documents that are stored using any of the above encodings. When an IDL application reads string data from the XML document using either the SAX or DOM parser, the string data is transcoded from the document's encoding into the encoding appropriate for IDL string variables. In order to read the string data correctly, the XML string data must be mappable into an IDL string. Since IDL strings use 1-byte characters, the XML strings must be transcodable into strings that use 1 byte per character. Further, they must be transcodable into strings that use the current character encoding. The IDL XML parsers may return an empty string if the XML string data cannot be converted into an IDL string.

──────────────────────────────────────────────────────────────────

It is beyond the scope of this manual to describe XML in detail. Numerous third-party books and electronic resources are available. The following texts may be useful:

- http://www.w3.org — information about many web standards, including XML related technologies.

- http://www.w3schools.com — tutorials on all manner of XML-related topics.

- http://www.saxproject.org — information about the Simple API for XML, the event-based XML parsing technology used by IDL.

- Brownell, David. *SAX2*. O'Reilly & Associates, 2002. ISBN: 0-596-00237-8.

- Harold, Eliotte Rusty. *XML Bible*. IDG Books Worldwide, 1999. ISBN: 0-7645-3236-7

# About XML Parsers

There are two basic types of parsers for XML data:

- Tree-based parsers
- Event-based parsers.

## Tree-Based Parsers

Tree-based parsers map an XML document into a tree structure in memory, allowing you to select elements by navigating through the tree. This type of parser is generally based on the Document Object Model (DOM) and the tree is often referred to as a DOM tree. The IDLffXMLDOM object classes implement a tree-based parser; for more information, see Chapter 21, "Using the XML DOM Object Classes".

Tree-based parsers are especially useful when the XML data file being parsed is relatively small. Having access to the entire data set at one time can be convenient and makes processing data based on multiple data values stored in the tree easy. However, if the tree structure is larger than will fit in physical memory or if the data must be converted into a new (local) data structure before use, then tree-based parsers can be slow and cumbersome.

## Event-Based Parsers

Event-based parsers read the XML document sequentially and report parsing events (such as the start or end of an element) as they occur, without building an internal representation of the data structure. The most common examples of event-based XML parsers use the Simple API for XML (SAX), and are often referred to as a SAX parsers.

Event-based parsers allow the programmer to write *callback routines* that perform an appropriate action in response to an event reported by the parser. Using an event-based parser, you can parse very large data files and create application-specific data structures. The IDLffXMLSAX object class implements an event-based parser based on the SAX version 2 API.

# Using the XML Parser

IDL's XML parser object class (IDLffXMLSAX) implements a SAX 2 event-based parser. The object's methods are a set of *callback routines* that are called automatically when the parser encounters different constituents of an XML document. For example, when the parser encounters the beginning of an XML element, it calls the StartElement method. When the StartElement method returns, the parser continues.

The IDLffXMLSAX object's methods are completely generic. As provided, they do nothing with the items encountered in the XML file. To use the parser object to read data from an XML file, you *must* write a subclass of the IDLffXMLSAX class, overriding the superclass's methods to accomplish your objectives. This requirement that you subclass the object makes the IDLffXMLSAX class unlike any other object class supplied by IDL.

For a detailed discussion of IDL object classes, subclassing, and method overriding, see Chapter 13, "Creating Custom Objects in IDL" (*Object Programming*). For a description of the parser object class and its methods, see "IDLffXMLSAX" (*IDL Reference Guide*).

## Subclassing the IDLffXMLSAX Object Class

Writing a subclass of the IDLffXMLSAX object class is similar to writing a subclass of any of IDL's other object classes. The basic steps are:

1.  Define a class structure for your subclass, inheriting from the IDLffXMLSAX object class.

2.  Write methods to override the IDLffXMLSAX object class methods as necessary.

3.  Write additional methods required for your application.

4.  Create a class definition routine for your XML parser object.

Let's look at these steps individually:

### Define a Class Structure

Every object class has a unique class structure that defines the instance data contained in the object. (See "Creating an Object Class Structure" (Chapter 13, *Object Programming*) for details.) When writing your own parser object (a subclass of the IDLffXMLSAX object), you must first determine what instance data you need your parser object to contain, and define a class structure accordingly.

**Note**

Your parser object's class structure must inherit from the IDLffXMLSAX class structure. See "Inheritance" (Chapter 13, *Object Programming*) for details.

For example, suppose you want to use your parser to extract an array of data from an XML file. You might choose to define your class structure to include an IDL pointer that will contain the data array. For this case, your class structure definition might look something like

```
void = {myParser, INHERITS IDLffXMLSAX, ptr:PTR_NEW()}
```

Within your subclass's methods, this data structure will always be available via the implicit `self` argument (see "Creating Custom Object Method Routines" (Chapter 13, *Object Programming*) for details). Setting the value of `self.ptr` within a method routine sets the instance data of the object.

In most cases, your class structure definition will be included in a routine that does *Automatic Structure Definition* (see "Automatic Class Structure Definition" (Chapter 13, *Object Programming*) for details).

## Override Superclass Methods

For your XML parser to do any work, you must override the generic methods of the IDLffXMLSAX object class. Overriding a method is as simple as defining a method routine with the same name as the superclass's method. When your parser encounters an item in the parsed XML file that triggers one of the IDLffXMLSAX methods, it will look first for a method of the same name in the definition of your subclass of the IDLffXMLSAX object class. See "Method Overriding" (Chapter 13, *Object Programming*) for details.

For example, suppose you want your parser to print out the element name of each XML element it encounters to IDL's output. You could override the `StartElement` method of the IDLffXMLSAX class as follows:

```
PRO myParser::StartElement, URI, Local, Name

PRINT, Name

END
```

**Note**

The new method must take the same parameters as the overridden method.

When your parser encounters the beginning of an XML element, it will look for a method named `StartElement` and call that method with the parameters specified

for the IDLffXMLSAX::StartElement method. Since your subclass's StartElement method is found before the superclass's StartElement method, your method is used.

**Note** ───────────────────────────────────────────────────────

You do not necessarily need to override *all* of the IDLffXMLSAX object methods. Depending on your application, it may be sufficient to override four or five of the superclass's methods. See the parser definitions later in this chapter for examples.

───────────────────────────────────────────────────────────────

Overriding the IDLffXMLSAX methods is the heart of writing your own XML parser. To write an efficient parser, you will need detailed knowledge of the structure of the XML file you want to parse.

See "Example: Reading Data Into an Array" on page 493 and "Example: Reading Data Into Structures" on page 500 for examples of how to work with parsed XML data and return the data in IDL variables.

## Write Additional Methods

Depending on your application, you may need to write additional object methods to work with the instance data retrieved from the parsed XML file. Like the overridden object methods, any new methods you write have access to the object's instance data via the implicit `self` parameter.

## Create a Class Definition Routine

If you combine your class definition routine with your class's method routines in a file, you can use IDL's *Automatic Structure Definition* feature to automatically compile the class routines when an instance of your class is created via the OBJ_NEW function. Keep the following in mind when creating the `.pro` file that will contain the definition of your class structure and method routines:

- The routine that creates your class structure should be named with the characters "__define" appended to the end of the class name. For example, if your parser object class is named "myParser" and its class structure is the one described in "Define a Class Structure" on page 488, the routine definition would be:

```
PRO myParser__define

void = {myParser, INHERITS IDLffXMLSAX, ptr:PTR_NEW()}

END
```

- The `.pro` file should be named after the class structure definition routine. In this case, the name would be `myParser__define.pro`.

• The class structure definition routine should be the last routine in the .pro file.

# Using Your Parser

Once you have written the class definition routine for your parser, you are ready to parse an XML file. The process is straightforward:

1. Create an instance of your parser object.

2. Call the ParseFile method on your object instance with the name of an XML file as the parameter.

For example, if your parser object is named myParser and the object class definition file is named myParser__define.pro, you could use the following IDL statements:

```
xmlFile = OBJ_NEW('myParser')
xmlFile->ParseFile, 'data.xml'
```

The first statement creates a new XML parser based on your class definition and places a reference to the parser object in the variable xmlFile. The second statement calls the ParseFile method on that object with the filename data.xml.

What happens next depends on your application. If your object definition stores values from the parsed file in the object's instance data, you will need some way to retrieve the values into IDL variables that are accessible outside the object. See "Example: Reading Data Into an Array" on page 493 and "Example: Reading Data Into Structures" on page 500 for examples that return data variables that are accessible to other routines.

# Validation

An XML document is said to be *valid* if it adheres to a set of constraints set forth in either a Document Type Definition (DTD) or an XML schema. Both DTDs and schemas define which elements can be included in an XML file and what values those elements can assume. XML schemas are a newer technology that is designed to replace and be more robust than DTDs. In working with existing XML files, you are likely to encounter both types of validation mechanisms.

Ensuring that a file contains valid XML helps in writing an efficient parsing mechanism. For example, if your validation method specifies that element B can only occur inside element A, and the XML document you are parsing is known to be valid, then your parser can assume that if it encounters element B it is inside element A.

The IDLffXMLSAX parser object can check an XML document using either validation mechanism, depending on whether a DTD or a schema definition is present. By default, if either is present, the parser will attempt to validate the XML document. See SCHEMA_CHECKING and VALIDATION_MODE under "IDLffXMLSAX Properties" (*IDL Reference Guide*) for details.

# Example: Reading Data Into an Array

This example subclasses the IDLffXMLSAX parser object class to create an object class named `xml_to_array`. The `xml_to_array` object class is designed to read numerical values from an XML file with the following structure:

```
<array>
  <number>0</number>
  <number>1</number>
  ...
</array>
```

and place those values into an IDL array variable.

**Note**
This example is a very simple example. It is designed to illustrate how an event-based XML parser is constructed using the IDLffXMLSAX object class. An application that reads real data from an XML file will most likely be quite a bit more complicated.

## Creating the xml_to_array Object Class

In order to read the XML file and return an array variable, we will need to create an object class definition that inherits from the IDLffXMLSAX object class, and override the following superclass methods: `Init`, `Cleanup`, `StartDocument`, `Characters`, `StartElement`, and `EndElement`. Since this example does not retrieve data using any of the other IDLffXMLSAX methods, we do not need to override those methods. In addition, we will create a new method that allows us to retrieve the array data from the object instance data.

**Example Code**
This example is included in the file `xml_to_array__define.pro` in the `examples/doc/file_io` subdirectory of the IDL distribution. Run the example procedure by entering `xml_to_array__define` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT xml_to_array__define.pro`.

### Object Class Definition

The following routine is the definition of the `xml_to_array` object class:

```
PRO xml_to_array__define
```

```
void = {xml_to_array, $
  INHERITS IDLffXMLSAX, $
  charBuffer:'', $
  pArray:PTR_NEW()}
END
```

The following items should be considered when defining this class structure:

- The structure definition uses the INHERITS keyword to inherit the object class structure and methods of the IDLffXMLSAX object.

- The `charBuffer` structure field is set equal to an empty string.

- The `pArray` structure field is set equal to an IDL pointer. We will use this pointer to store the numerical array data we retrieve.

- The routine name is created by adding the string "__define" (note the *two* underscore characters) to the class name.

Why do we store the array data in a pointer variable? Because the fields of a named structure (`xml_to_array`, in this case) must always contain the same type of data as when that structure was defined. Since we want to be able to add values to the data array as we parse the XML file, we will need to extend the array with each new value. If we began by defining the size of the array in the structure variable, we would not be able to extend the array. By holding the data array in a pointer, we can extend the array without changing the format of the `xml_to_array` object class structure.

**Note**

Although we describe this routine first here, the `xml_to_array__define` routine must be the *last* routine in the `xml_to_array__define.pro` file.

## Init Method

The `Init` method is called when the an `xml_to_array` parser object is created by a call to OBJ_NEW. The following routine is the definition of the `Init` method:

```
FUNCTION xml_to_array::Init
  self.pArray = PTR_NEW(/ALLOCATE_HEAP)
  RETURN, self->IDLffxmlsax::Init()
END
```

We do two things in this method:

- We initialize the pointer in the pArray field of the class structure variable.

    **Note** ──────────────────────────────

    Within a method, we can refer to the class structure variable with the implicit parameter self. Remember that self is actually a reference to the xml_to_array object instance.
    ────────────────────────────────────────

- The return value from this function is the return value of the superclass's Init method, called on the self object reference.

**Note** ────────────────────────────────
The initialization task (setting the value of the pArray field) is performed before calling the superclass's Init method.
────────────────────────────────────────

See "IDLffXMLSAX::Init" (*IDL Reference Guide*) for details on the method we are overriding.

## Cleanup Method

The Cleanup method is called when the xml_to_array parser object is destroyed by a call to OBJ_DESTROY. The following routine is the definition of the Cleanup method:

```
PRO xml_to_array::Cleanup

IF (PTR_VALID(self.pArray)) THEN PTR_FREE, self.pArray

self->IDLffXMLSAX::Cleanup

END
```

Here, we release the pArray pointer, if it exists, and call the superclass cleanup method.

See "IDLffXMLSAX::Cleanup" (*IDL Reference Guide*) for details on the method we are overriding.

## Characters Method

The Characters method is called when the xml_to_array parser encounters character data inside an element. The following routine is the definition of the Characters method:

```
PRO xml_to_array::characters, data
```

```
self.charBuffer = self.charBuffer + data

END
```

As it parses the character data in an element, the parser will read characters until it reaches the end of the text section. Here, we simply add the current characters to the charBuffer field of the object's instance data structure.

See "IDLffXMLSAX::Characters" (*IDL Reference Guide*) for details on the method we are overriding.

## StartDocument Method

The StartDocument method is called when the xml_to_array parser encounters the beginning of the XML document. The following routine is the definition of the StartDocument method:

```
PRO xml_to_array::StartDocument

IF (N_ELEMENTS(*self.pArray) GT 0) THEN $
    void = TEMPORARY(*self.pArray)

END
```

Here, we check to see if the array pointed at by the pArray pointer contains any data. Since we are just beginning to parse the XML document at this point, it should not contain any data. If data is present, we reinitialize the array using the TEMPORARY function.

**Note**
Since pArray is a pointer, we must use dereferencing syntax to refer to the array.

See "IDLffXMLSAX::StartDocument" (*IDL Reference Guide*) for details on the method we are overriding.

## StartElement Method

The StartElement method is called when the xml_to_array parser encounters the beginning of an XML element. The following routine is the definition of the StartElement method:

```
PRO xml_to_array::startElement, URI, local, strName, attr, value

CASE strName OF
    "array": BEGIN
        IF (N_ELEMENTS(*self.pArray) GT 0) THEN $
        void = TEMPORARY(*self.pArray);; clear out memory
    END
```

```
      "number" : BEGIN
          self.charBuffer = ''
      END
   ENDCASE

   END
```

Here, we first check the name of the element we have encountered, and use a CASE
statement to branch based on the element name:

- If the element is an <array> element, we check to see if the array pointed at
  by the pArray pointer is empty. Since we are just beginning to read the array
  data at this point, there should be no data. If data already exists, we reinitialize
  the array using the TEMPORARY function.

- If the element is a <number> element, we reinitialize the charBuffer field.
  Since we are just beginning to read the number data, nothing should be in the
  buffer.

See "IDLffXMLSAX::StartElement" (*IDL Reference Guide*) for details on the
method we are overriding.

## EndElement Method

The EndElement method is called when the xml_to_array parser encounters the
end of an XML element. The following routine is the definition of the EndElement
method:

```
   PRO xml_to_array::EndElement, URI, Local, strName

   CASE strName OF
      "array":
      "number": BEGIN
          idata = FIX(self.charBuffer);
          IF (N_ELEMENTS(*self.pArray) EQ 0) THEN $
             *self.pArray = iData $
          ELSE $
             *self.pArray = [*self.pArray,iData]
      END
   ENDCASE

   END
```

As with the StartElement method, we first check the name of the element we have
encountered, and use a CASE statement to branch based on the element name:

- If the element is an <array> element, we do nothing.

- If the element is a `<number>` element, we must get the data stored in the `charBuffer` field of the instance data structure and place it in the array:

  - First, we convert the string data in the `charBuffer` into an IDL integer.

  - Next, we check to see if the array pointed at by `pArray` is empty. If it is empty, we simply set the array equal to the data value we retrieved from the `charBuffer`.

  - If the array pointed at by `pArray` is not empty, we redefine the array to include the new data retrieved from the `charBuffer`.

See "IDLffXMLSAX::EndElement" (*IDL Reference Guide*) for details on the method we are overriding.

**Note** ─────────────────────────────────────────────────────────

In both the `StartElement` and `EndElement` methods, we rely on the validity of the XML data file. Our CASE statements only need to handle the element types described in the XML file's DTD or schema (in this case, the only elements are `<array>` and `<number>`). We do not need an ELSE clause in the CASE statement. If an unknown element is found in the XML file, the parser will report a validation error.

─────────────────────────────────────────────────────────────────

## GetArray Method

The `GetArray` method allows us to retrieve the array data stored in the `pArray` pointer variable. The following routine is the definition of the `GetArray` method:

```
FUNCTION xml_to_array::GetArray

IF (N_ELEMENTS(*self.pArray) GT 0) THEN $
   RETURN, *self.pArray $
ELSE RETURN , -1

END
```

Here, we check to see whether the array pointed at by `pArray` contains any data. If it does contain data, we return the array. If the array contains no data, we return the value `-1`.

# Using the xml_to_array Parser

To see the xml_to_array parser in action, you can parse the file num_array.xml, found in the examples/data subdirectory of the IDL distribution. This num_array.xml file contains the fragment of XML like the one shown in the beginning of this section, and includes 20 extra <number> elements. The num_array.xml file also includes a DTD describing the structure of the file.

Enter the following statements at the IDL command line:

```
xmlObj = OBJ_NEW('xml_to_array')
xmlFile = FILEPATH('num_array.xml', $
    SUBDIRECTORY = ['examples', 'data'])
xmlObj->ParseFile, xmlFile
myArray = xmlObj->GetArray()
OBJ_DESTROY, xmlObj
HELP, myArray
PRINT, myArray
```

IDL prints:

```
MYARRAY          INT      = Array[20]
   0    1    2    3    4    5    6    7    8    9   10   11
  12   13   14   15   16   17   18   19
```

# Example: Reading Data Into Structures

This example subclasses the IDLffXMLSAX parser object class to create an object class named `xml_to_struct`. The `xml_to_struct` object class is designed to read data from an XML file with the following structure:

```
<Solar_System>
   <Planet NAME='Mercury'>
      <Orbit UNITS='kilometers' TYPE='ulong64'>579100000</Orbit>
      <Period UNITS='days' TYPE='float'>87.97</Period>
      <Satellites TYPE='int'>0</Satellites>
   </Planet>
 ...
</Solar_System>
```

and place those values into an IDL array containing one structure variable for each `<Planet>` element. We use a structure variable for each `<Planet>` element so we can capture data of several data types in a single place.

**Note** ────────────────────────────────────────────────

While this example is more complicated than the previous example, it is still rather simple. It is designed to illustrate a method whereby more complex XML data structures can be represented in IDL.

────────────────────────────────────────────────────────

## Creating the xml_to_struct Object Class

To read the XML file and return a structure variable, we will need to create an object class definition that inherits from the IDLffXMLSAX object class, and override the following superclass methods: `Init`, `Characters`, `StartElement`, and `EndElement`. Since this example does not retrieve data using any of the other IDLffXMLSAX methods, we do not need to override those methods. In addition, we will create a new method that allows us to retrieve the structure data from the object instance data.

Notice that the elements of the XML data file include *attributes*. While we will retrieve and use some of the attribute data from the file, we will ignore some of it.

**Note** ────────────────────────────────────────────────

When parsing an XML data file, you can pick and choose the data you wish to pull into IDL. This ability to selectively retrieve data from the XML file is one of the great advantages of an event-based parser over a tree-based parser.

────────────────────────────────────────────────────────

**Example Code**

This example is included in the file `xml_to_struct__define.pro` in the `examples/doc/file_io` subdirectory of the IDL distribution. Run the example procedure by entering `xml_to_struct__define` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT xml_to_struct__define.pro`.

## Object Class Definition

The following routine is the definition of the `xml_to_struct` object class:

```
PRO xml_to_struct__define

void = {PLANET, NAME: "", Orbit: 0ull, period:0.0, Moons:0}
void = {xml_to_struct, $
   INHERITS IDLffXMLSAX, $
   CharBuffer:"", $
   planetNum:0, $
   currentPlanet:{PLANET}, $
   Planets : MAKE_ARRAY(9, VALUE = {PLANET})}

END
```

The following items should be considered when defining this class structure:

- Before creating the object class structure, we define a structure named PLANET. We will use the PLANET structure to store data from the `<Planet>` elements of the XML file.

- The object class structure definition uses the INHERITS keyword to inherit the object class structure and methods of the IDLffXMLSAX object.

- The `charBuffer` structure field is set equal to a string value. We will use this field to accumulate character data stored in XML elements.

- The `planetNum` structure field is set equal to an integer value. We will use this field to keep track of which array element we are currently populating.

- The `currentPlanet` structure field is set equal to a PLANET structure.

- The `Planets` structure field is set equal to a nine-element array of PLANET structures.

- The routine name is created by adding the string "__define" (note the *two* underscore characters) to the class name.

We have explicitly defined our Planets structure field as a nine-element array of PLANET structures, which we can do because we know exactly how many <Planet> elements will be read from our XML file. Specifying the exact size of the data array in the class structure definition is very efficient (since we create the array only once) and eliminates the need to free the pointer in the Cleanup method. However, it has the following consequences:

- We must explicitly keep track of the index of the array element we are populating, and increment it after we have finished with a given element (see the EndElement method below).

- We must know in advance how many elements the array will hold. If the size of the final array is unknown, it is more efficient to use a pointer to an array, as we did in the previous example, and allow the array to grow as elements are added. See "Building Complex Data Structures" on page 507 for additional discussion of ways to configure the instance data structure.

**Note** ─────────────────────────────────────────────────────────

Although we describe this routine here first, the xml_to_struct__define routine must be the last routine in the xml_to_struct__define.pro file.

─────────────────────────────────────────────────────────────────

## Init Method

The Init method is called when the an xml_to_struct parser object is created by a call to OBJ_NEW. The following routine is the definition of the Init method:

```
FUNCTION xml_to_struct::Init

self.planetNum = 0
RETURN, self->IDLffXMLSAX::Init()

END
```

We do two things in this method:

- We initialize the planetNum field with the value of zero. We will increment this value as we populate the Planets array.

    **Note** ─────────────────────────────────────────────────────

    Within a method, we can refer to the class structure variable with the implicit parameter self. Remember self is actually a reference to the xml_to_struct object instance.

    ─────────────────────────────────────────────────────────────

- The return value from this function is the return value of the superclass's Init method, called on the self object reference.

**Note** ─────────────────────────────────

We perform our own initialization task (setting the value of the planetNum field) before calling the superclass's Init method.

─────────────────────────────────────────

See "IDLffXMLSAX::Init" (*IDL Reference Guide*) for details on the method we are overriding.

## Characters Method

The Characters method is called when the xml_to_struct parser encounters character data inside an element. The following routine is the definition of the Characters method:

```
PRO xml_to_struct::characters, data

self.charBuffer = self.charBuffer + data

END
```

As it parses the character data in an element, the parser will read characters until it reaches the end of the text section. Here, we simply add the current characters to the charBuffer field of the object's instance data structure.

See "IDLffXMLSAX::Characters" (*IDL Reference Guide*) for details on the method we are overriding.

## StartElement Method

The StartElement method is called when the xml_to_struct parser encounters the beginning of an XML element. The following routine is the definition of the StartElement method:

```
PRO xml_to_struct::startElement, URI, local, strName, attrName,
attrValue

CASE strName OF
   "Solar_System":   ; Do nothing
   "Planet" : BEGIN
      self.currentPlanet = {PLANET, "", 0ull, 0.0, 0}
      self.currentPlanet.Name = attrValue[0]
   END
   "Orbit" : self.charBuffer = ''
   "Period" : self.charBuffer = ''
   "Moons" : self.charBuffer = ''
ENDCASE

END
```

Here, we first check the name of the element we have encountered, and use a CASE statement to branch based on the element name:

- If the element is a `<Solar_System>` element, we do nothing.

- If the element is a `<Planet>` element, we do the following things:

  - Set the value of the currentPlanet field of the self instance data structure equal to a PLANET structure, setting the values of the structure fields to zero values.

  - Set the value of the Name field of the PLANET structure held in the currentPlanet field equal to the value of the Name attribute of the element. This field contains the name of the planet whose data we are reading.

- If the element is an `<Orbit>`, `<Period>`, or `<Moons>` element, we reinitialize the value of the charBuffer field of the self instance data structure.

See "IDLffXMLSAX::StartElement" (*IDL Reference Guide*) for details on the method we are overriding.

## EndElement Method

The EndElement method is called when the xml_to_struct parser encounters the end of an XML element. The following routine is the definition of the EndElement method:

```
PRO xml_to_struct::EndElement, URI, Local, strName

CASE strName of
   "Solar_System":
   "Planet": BEGIN
      self.Planets[self.planetNum] = self.currentPlanet
      self.planetNum = self.planetNum + 1
   END
   "Orbit" : self.currentPlanet.Orbit = self.charBuffer
   "Period" : self.currentPlanet.Period = self.charBuffer
   "Moons" : self.currentPlanet.Moons= self.charBuffer
ENDCASE

END
```

As with the StartElement method, we first check the name of the element we have encountered, and use a CASE statement to branch based on the element name:

- If the element is a <Solar_System> element, we do nothing.

- If the element is a <Planet> element, we set the element of the Planets array specified by planetNum equal to the PLANET structure contained in currentPlanet. Then, we increment the planetNum counter.

- If the element is an <Orbit>, <Period>, or <Satellites> element, we place the value in the charBuffer field into the appropriate field within the PLANET structure contained in currentPlanet.

See "IDLffXMLSAX::EndElement" (*IDL Reference Guide*) for details on the method we are overriding.

**Note**

In both the StartElement and EndElement methods, we rely on the validity of the XML data file. Our CASE statements only need to handle the element types described in the XML file's DTD or schema. We do not need an ELSE clause in the CASE statement. If an unknown element is found in the XML file, the parser will report a validation error.

## GetArray Method

The GetArray method allows us to retrieve the array of structures stored in the Planets variable. The following routine is the definition of the GetArray method:

```
FUNCTION xml_to_struct::GetArray

IF (self.planetNum EQ 0) THEN $
   RETURN, -1 $
ELSE RETURN, self.Planets[0:self.planetNum-1]

END
```

Here, we check to see whether the planetNum counter has been incremented. If it has been incremented, we return as the number of array elements specified by the counter. If the counter has not been incremented (indicating that no data has been stored in the array), we return the value -1.

# Using the xml_to_struct Parser

To see the xml_to_struct parser in action, you can parse the file planets.xml, found in the examples/data subdirectory of the IDL distribution. The planets.xml file contains the fragment of XML like the one shown at the beginning of this section, and includes a <Planet> element for each planet in the solar system. The planets.xml file also includes a DTD describing the structure of the file.

Enter the following statements at the IDL command line:

```
xmlObj = OBJ_NEW('xml_to_struct')
xmlFile = FILEPATH('planets.xml', $
    SUBDIRECTORY = ['examples', 'data'])
xmlObj->ParseFile, xmlFile
planets = xmlObj->GetArray()
OBJ_DESTROY, xmlObj
```

The variable planets now holds an array of PLANET structures, one for each planet. To print the number of moons for each planet, you could use the following IDL statement:

```
FOR i = 0, (N_ELEMENTS(planets.Name) - 1) DO $
    PRINT, planets[i].Name, planets[i].Moons, $
    FORMAT = '(A7, " has ", I2, " moons")'
```

IDL prints:

```
Mercury has  0 moons
Venus   has  0 moons
Earth   has  1 moons
Mars    has  2 moons
Jupiter has 16 moons
Saturn  has 18 moons
Uranus  has 21 moons
Neptune has  8 moons
Pluto   has  1 moons
```

To view all the information about the planet Mars, you could use the following IDL statement:

```
HELP, planets[3], /STRUCTURE
```

IDL prints:

```
** Structure PLANET, 4 tags, length=32, data length=26:
   NAME        STRING      'Mars'
   ORBIT       ULONG64     227940000
   PERIOD      FLOAT       686.980
   MOONS       INT         2
```

# Building Complex Data Structures

Few limitations exist regarding the complexity of the data structures that can be represented in an XML data file. Writing a parser to read data from such complex structures into IDL can be a challenge. If you are designing a parser to read a very complex or deeply nested XML file, keep the following concepts in mind.

## Use Dynamically Sized Arrays if Necessary

If you don't know the final size of your data array, or if the size of the array will change, store the data array in an IDL pointer in the instance data structure. This technique allows you to change the size of the data array without changing the definition of the instance data structure. The downside of extending the data array in this manner is performance. Each time the array is extended, IDL must hold two copies of the entire array in memory. If the array becomes large, this duplication can cause performance problems.

In "Example: Reading Data Into an Array" on page 493, we extended our data array as we added each element despite the fact that we knew the number of data elements. We used a pointer to illustrate the technique, and to make it clear that if you use pointers to store your instance data, you must free the pointers in your subclass's `Cleanup` method.

## Use Fixed-Size Arrays When Possible

If you will be building a large data array, and you know in advance how many elements it will contain, create the array when defining the class data structure and use array indexing to place data in the appropriate elements. Using a fixed-size array eliminates the need to copy the full array each time it is extended, and can lead to noticeable performance improvements when large arrays are involved.

In "Example: Reading Data Into Structures" on page 500, we illustrated the technique of using a pre-defined array to store our instance data.

## Using Nested Structures

If your data structure is complex, you may be inclined to represent your data as a set of nested IDL structure variables. While nesting structure variables can help you create a data structure that emulates the structure of your XML file, deeply nested structures can make your code more difficult to create and maintain. Consider storing data in several arrays of structures rather than a single, deeply-nested structure.

If you have a good reason to create nested structures, and also need to extend them dynamically, you should use the CREATE_STRUCT function.

The same caveats apply to extending a structure with CREATE_STRUCT as apply to extending an array. With large datasets, the process of duplicating the structures may cause performance problems.

# Chapter 21
# Using the XML DOM Object Classes

The following topics are covered in this chapter:

# About the Document Object Model

The Document Object Model (DOM) describes the content of XML data in the form of a document object, which contains other objects that describe the various data elements of the XML document. The DOM also specifies an interface for interacting with the objects in the model. This is the interface exposed to the IDL user.

**Note** ─────────────────────────────────────────────────────────────────
For more information on XML, see "About XML" on page 486.
──────────────────────────────────────────────────────────────────────────

## When to Use the DOM

There are two basic types of parsers for XML data: object-based and event-based. The DOM is object-based and as such has advantages in certain situations over an event-based parser such as SAX. In general, use the DOM:

- To access an XML document in any order (SAX must parse in file order)

- To write to a file (SAX does not support modifying or creating XML data)

For more information on the difference between the two parsers, see "About XML Parsers" on page 487.

## About the DOM Structure

Here is an example of an XML file that is used in an application to define a weather-monitoring plug-in component:

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin type="tab-iframe">
   <name>Weather.com Radar Image [DEN]</name>
   <description>600 mile Doppler radar image for DEN</description>
   <version>1.0</version>
   <tab>
      <icon>weather.gif</icon>
      <tooltip>DEN Doppler radar image</tooltip>
   </tab>
</plugin>
```

The contents of this file constitute an XML document. When you want to work with this data, you can use IDL to load the file, parse it, and store it in memory in DOM format. The sample file listed above is stored in the DOM structure as shown in Figure 21-1.
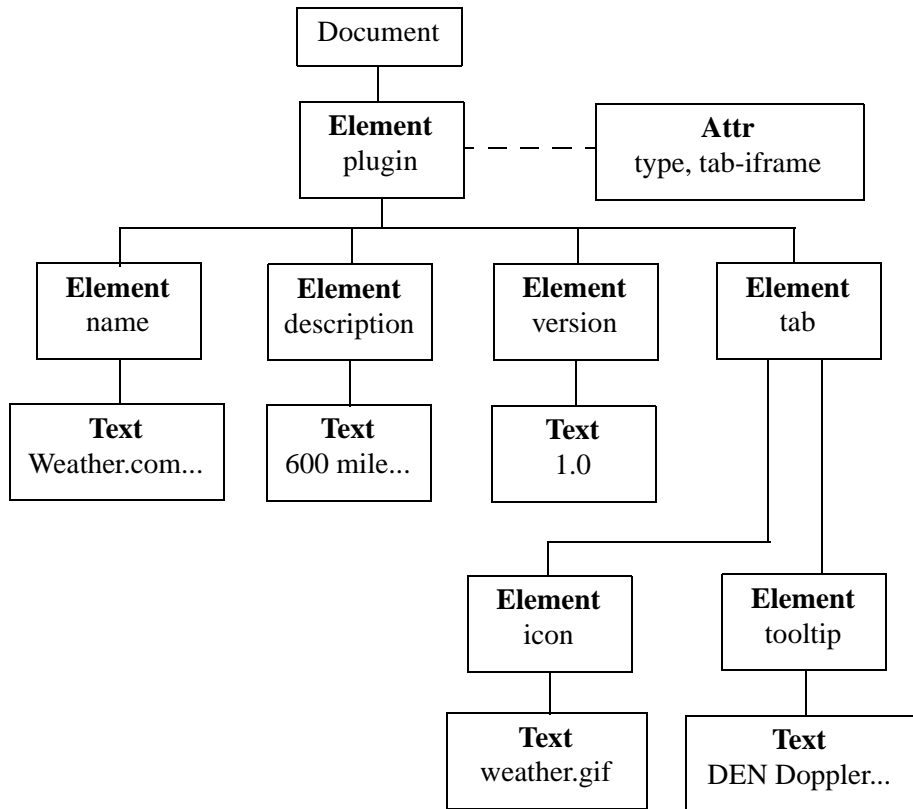
*Figure 21-1: XML DOM Tree Structure: Plug-in Example*

The DOM structure is a tree of nodes, where each node is represented as a box in the figure. The type of each node is in **boldface**. The contents of the node are in normal type.

Note that whitespace and newline characters can appear in this tree as text nodes, but are omitted in this picture for clarity. It is important to keep this in mind when exploring the DOM tree. There are parsing options available that can prevent the creation of ignorable-whitespace nodes (see "Working with Whitespace" on page 525).

The attribute node (Attr) is not actually a child of the element node, but is still associated with it, as indicated by the dotted line.

# How IDL Uses the DOM Structure

To access the XML data in the structure, you need to create a set of IDL objects that correspond to the portion of the DOM tree in which you are interested. You use the following process to create the DOM tree and the corresponding IDL objects:

1.  Create an IDLffXMLDOMDocument object.

2.  Load the XML file. This step parses the XML data from the file and creates the DOM tree in memory.

3.  Use the IDLffXMLDOMDocument object to create IDLffXMLDOM objects that essentially mirror portions of the DOM tree, as shown in Figure 21-2.

You then use the IDLffXMLDOM objects to access the actual XML data contained in the DOM tree.
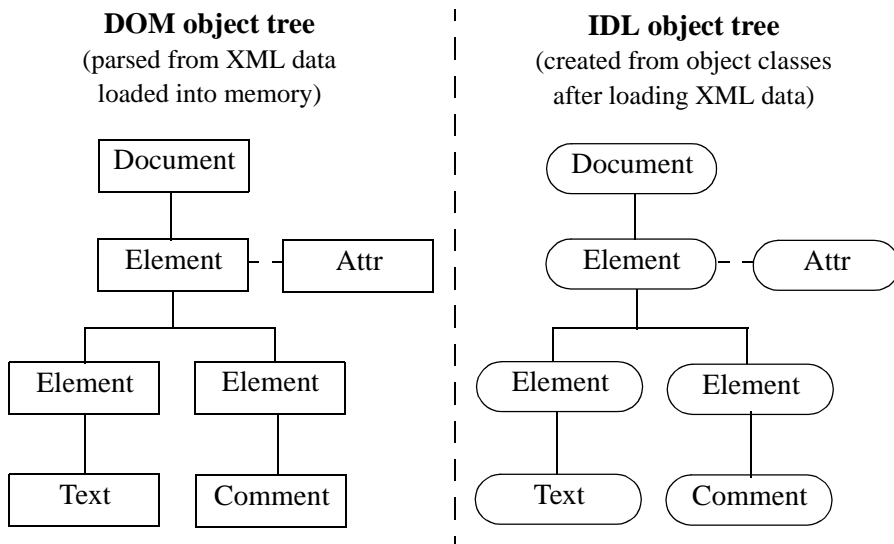


*Figure 21-2: The DOM and IDL Trees*

The creation and destruction of the IDL objects do not alter the DOM structure. There are explicit methods for modifying the DOM structure. The IDL objects are merely access objects that are used to manipulate the DOM tree nodes.

# About the XML DOM Object Classes

The IDL XML DOM support is provided by a set of IDL object classes, all starting with *IDLffXMLDOM*. These classes provide access to the XML document via the DOM. The IDLffXMLDOM objects do not in themselves maintain a copy of the document data. Instead, they provide access to the data stored in the DOM document structure.

## IDLffXMLDOMNode Class Hierarchy

One of the key object classes is IDLffXMLDOMNode. Because it is an abstract class, you will never create an instance of this class. The node is the basic DOM data structure used to map each DOM data element. The nodes are organized in a classic tree structure, according to the layout of the data in the document.

The following classes are derived from IDLffXMLDOMNode, where each class is named *IDLffXML<node type>* (e.g., IDLffXMLDOMAttr):
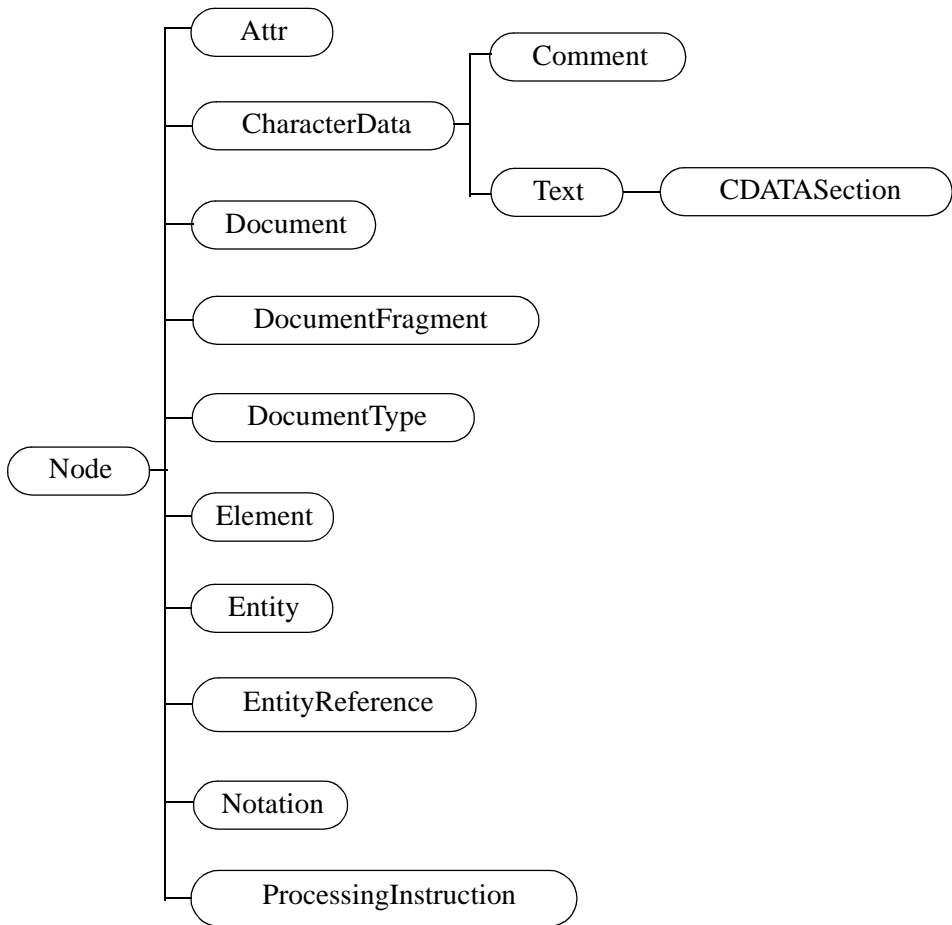
*Figure 21-3: The IDLffXMLDOMNode Class Hierarchy*

These classes represent the data that can be stored in an XML document. Except for the IDLffXMLDOMDocument class, you do not instantiate any of them directly. To begin working with the IDL XML DOM interface, you use the OBJ_NEW function to create an IDLffXMLDOMDocument object. You then use this object to browse and modify the document. This document object also creates objects using the derived classes to give you access to the various parts of the document.

For example:

```
oChild = oMyDOMDocument->GetFirstChild()
```

creates an IDL object of one of the node types, depending on what the first child in your document actually is. The newly created IDL object refers to the first child node of the document and does not modify the document in any way.

You then use the `oChild` object's methods to get data from the node, modify the node, or find another node.

Because of the class hierarchy, all the methods in a superclass are available to its subclasses. For example, to determine which methods are available for use by an object of the IDLffXMLDOMText class, you would have to look at the methods belonging to the IDLffXMLDOMText, IDLffXMLDOMCharacterData, and IDLffXMLDOMNode classes.

**Note**

The IDLffXMLDOMCharacterData class is a special abstract class that provides character-handling facilities for its subclasses. You will never create an instance of this class.

# IDLffXMLDOM Object Helper Classes

IDL provides a set of other classes to assist you in navigating the DOM tree. These classes are:

- **•** IDLffXMLDOMNodeIterator — navigates in a depth-first, document-order traversal.

- **•** IDLffXMLDOMTreeWalker — navigates in a tree-walking traversal.

- **•** IDLffXMLDOMNodeList — contains a list of children of a node. You can create node lists using the GetElementsByTagName and GetChildNodes methods, for example.

- **•** IDLffXMLDOMNamedNodeMap — contains a list of attributes from an element node that are looked up by attribute name.

The IDLffXMLDOMNodeIterator and IDLffXMLDOMTreeWalker classes do not contain lists used in tree traversal. Instead, they each operate by creating a node object for accessing a DOM node and then destroying that node object as the iterator or walker is moved to another DOM node. Conceptually, both node iterators and tree walkers are "current" node pointers into the DOM tree. For more information, see the classes' respective documentation in the *IDL Reference Guide*.

The IDLffXMLDOMNodeList and IDLffXMLDOMNamedNodeMap classes contain nodes that are subclasses of IDLffXMLDOMNode. Node lists and named node maps are active collections of nodes that are updated as the DOM tree is modified. That is, they are not static snapshots of a DOM tree in a given state; the list contents are modified as the DOM tree is modified. While this dynamic update is useful because you do not have to take specific action to update a list after modifying the tree, it can be confusing in some situations.

Suppose you want to delete all the children of an element node. The following code seems to make sense:

```
oList = oElement->GetChildNodes()
n = oList->GetLength()
FOR i=0, n-1 DO $
   oDeleted = oElement->RemoveChild(oList->Item(i))
```

This approach does not work as expected because after the first child is deleted, the list is updated so it contains one fewer object, and the indexes of all remaining objects are decremented by one. As the loop continues, some items are not deleted, and eventually an error occurs when the loop index *i* exceeds the length of the shortened list.

The following code performs the intended deletion, by changing the parameter to the Item method from *i* to 0:

```
oList = oElement->GetChildNodes()
n = oList->GetLength()
FOR i=0, n-1 DO $
   oDeleted = oElement->RemoveChild(oList->Item(0))
```

This code works because each time the first child is deleted, the list is automatically updated to place another object in the first position.

The following approach might be more appealing:

```
oList = oElement->GetChildNodes()
n = oList->GetLength()
FOR i=n-1, 0, -1 DO $
   oDeleted = oElement->RemoveChild(oList->Item(i))
```

This code works because it deletes items from the end of the list, rather than from the beginning.

## IDL Node Ownership

Whenever you create an IDLffXMLDOM node object with a method such as IDLffXMLDOMNode::GetFirstChild, you are also creating an ownership relationship between the created node object and the node object that created it.

Working from the previous plug-in example (see "About the DOM Structure" on page 510), suppose that you have an object reference, oName, to an instance of the IDLffXMLDOMElement class that refers to the first child of the plug-in node:

```
oName = oDocument->GetFirstChild()
```

Using oName, you can issue the following call:

```
oDescription = oName->GetNextSibling()
```

The description and name DOM nodes are siblings of each other in the DOM tree, as shown in Figure 21-2. The IDL object oDescription refers to the description node in the DOM tree, and the IDL object oName refers to the name node in the DOM tree. However, the oDescription object is owned by the oName object because oName created oDescription.

You might understand this relationship better by realizing that the parent/sibling relationships in the DOM tree reflect the DOM tree structure and that the ownership relationships among the IDL access objects are due to the creation of the IDL access objects. Because oName created oDescription, oName destroys oDescription when oName is destroyed, even though they refer to siblings in the DOM tree. Bear in mind that destroying these access objects does not affect the DOM tree itself.

This parent relationship among IDLffXMLDOM objects is useful for cleaning them up. Because all of the objects that might have been created during the exploration of a DOM tree are all ultimately descendants of an IDLffXMLDOMDocument node, simply destroying the document object is sufficient to clean up all the nodes. Unless you are concerned with cleaning up some access objects at a particular time (to save memory, for example), you can simply wait to clean them all up when you are finished with the data by destroying the IDLffXMLDOMDocument node.

To reduce memory requirements, you can destroy node objects that are no longer needed. For example, if you wanted to explore all the children of a given element oElement, you might use the following code:

```
oFirstChild = oElement->GetFirstChild()
oChild = oFirstChild
WHILE OBJ_VALID(oChild) DO BEGIN
    PRINT, oChild->GetNodeValue()
    oChild = oChild->GetNextSibling()
ENDWHILE
OBJ_DESTROY, oFirstChild
```

This approach works well because all the node objects created during the exploration of the children by the GetNextSibling method are destroyed when oFirstChild is destroyed. While it would seem that objects "lost" to the reassignment of oChild would not be accessible for destruction, the chain of oChild objects keeps track of

them and destroys them all when the head of the chain, saved in `oFirstChild`, is destroyed.

Trying to destroy node objects inside the loop as follows does not work as expected:

```
oChild = oElement->GetFirstChild()
WHILE OBJ_VALID(oChild) DO BEGIN
   PRINT, oChild->GetNodeValue()
   oNext = oChild->GetNextSibling()
   OBJ_DESTROY, oChild
   OChild = oNext
ENDWHILE
```

This code fails because when `oChild` is destroyed for the first time, it also destroys `oNext`, causing the loop to exit after the first iteration.

If there is a very large number of children, waiting until the end of the loop to destroy the list might be too inefficient. Using a node list, as in the following code, is an alternative:

```
oList = oElement->GetChildNodes()
n = oList->GetLength()
FOR i=0, n-1 DO BEGIN
   oChild = oList->Item(i)
   PRINT, oChild->GetNodeValue()
   OBJ_DESTROY, oChild
ENDFOR
OBJ_DESTROY, oList
```

Although `oList` requires some space to maintain the list, there is only one valid node connected to `oChild` in memory each time through the loop.

You can change the node deletion policy so that nodes created by a node are not deleted when the node is destroyed. This change lets the following code work properly:

```
oDocument->SetProperty, NODE_DESTRUCTION_POLICY=1
oChild = oElement->GetFirstChild()
WHILE OBJ_VALID(oChild) DO BEGIN
   PRINT, oChild->GetNodeValue()
   oNext = oChild->GetNextSibling()
   OBJ_DESTROY, oChild
   oChild = oNext
ENDWHILE
```

Now, the `OBJ_DESTROY` call no longer destroys the object to which `oNext` refers, and the loop proceeds as expected.

# Saving and Restoring IDLffXMLDOM Objects

IDL does not save IDLffXMLDOM objects in a SAVE file. If you restore a SAVE file that contains object references to IDLffXMLDOM objects, the object references are restored, but are set to null object references.

The IDLffXMLDOM objects are not saved because they contain state information for the external Xerces library. This state information is not available to IDL and cannot be restored. The contents of the XML file might also have changed, which would also make any saved state invalid.

It is recommended that applications either complete any DOM operations before saving their data in a SAVE file or reload the DOM document as part of restoring their state.

# Using the XML DOM Classes

Continuing from the weather plug-in example (see "About the DOM Structure" on page 510), this section describes how to use the IDL XLM DOM object classes, namely how to do the following actions:

- Load an XML document

- Read XML data from a document

- Modify existing XML data

- Create new XML data

- Destroy IDLffXMLDOM objects

## Loading an XML Document

Although the DOM tree structure is in memory after the XML file is loaded, you cannot directly access the data from IDL until you have created IDLffXMLDOM objects to access them. The DOM loads and parses the XML data into a tree structure, but you need to create a document object to access that data through a mirroring IDL tree structure.

To prepare the interface, load the document:

```
oDocument = OBJ_NEW('IDLffXMLDOMDocument')
oDocument->Load, FILENAME='sample.xml'
```

This code causes the DOM tree structure to be formed in memory. You could also perform the same action in one line:

```
oDocument = OBJ_NEW('IDLffXMLDOMDocument', FILENAME='sample.xml')
```

Be aware that either of these examples will discard an existing DOM tree referenced by `oDocument`. You can load and reload an XML file as often as desired, but each loading action will overwrite, not add to, the existing tree and remove its objects from memory.

**Tip** ─────────────────────────────────────────────────────────
You can read from and write to IDL variables rather than disk files, see "IDLffXMLDOMDocument::Init" (*IDL Reference Guide*) for more details.
─────────────────────────────────────────────────────────────────

# Reading XML Data

Suppose that you want to print the name of the plug-in. The plug-in element node is the first and only child of the document node. A document node can have only one element child node, which represents the containing element for the entire document (for comparison, consider that an HTML file has only one <HTML></HTML> pair). The name of the element node is the first element child of the plug-in element. There may be several ways to locate a desired piece of data using the IDL XML DOM classes. The following example illustrates one way to find the plug-in name.

First, access the first child of the document, which is the plug-in element:

```
oPlugin = oDocument->GetFirstChild()
```

The GetFirstChild method creates an IDLffXMLDOMElement node object and returns its object reference, which is stored in oPlugin.

Next, ask the plug-in for a list of all of its child element nodes. The oPlugin object creates an IDLffXMLDOMNodeList object and places all the child element nodes in the list. You could have asked for only the name element, but by asking for them all, you will have the other elements in the list in case you need to look at them later.

```
oNodeList = oPlugin->GetElementsByTagName('*')
```

You know from the design of the XML data, perhaps as defined in a DTD, that the name element must always be the first child of a plug-in element. You can access the name as follows:

```
oName = oNodeList->Item(0)
```

You also know that the name element can only contain a text node. Getting access to the text node lets you print the data that you want.

```
oNameText = oName->GetFirstChild()
PRINT, oNameText->GetNodeValue()
```

This command prints out:

```
Weather.com Radar Image [DEN]
```

Note that the oPlugin and the oName objects are of type IDLffXMLDOMElement, and the oNameText object is of type IDLffXMLDOMText. The oName and oNameText objects are created by the GetFirstChild and Item methods, using the object class that is appropriate for the type of data in the DOM tree. You used the GetElementsByTagName method to get the child elements of the plug-in, without having to sort through the whitespace text nodes that are present.

At this point, you have four IDL objects in addition to the root document object that give you access to only the portion of the DOM tree to which these objects

correspond. You can create additional objects to explore other parts of the tree and destroy objects for parts that you are no longer interested in.

# Modifying Existing Data

You can also modify XML data and write the result back out to a file.

```
oDocument = OBJ_NEW('IDLffXMLDOMDocument')
oDocument->Load, FILENAME='sample.xml'
oPlugin = oDocument->GetFirstChild()
oNodeList = oPlugin->GetElementsByTagName('*')
oName = oNodeList->Item(0)
oNameText = oName->GetFirstChild()
oNameText->SetNodeValue, 'Weather.com Radar Image [PDX]'
oDocument->Save, FILENAME='sample2.xml'
OBJ_DESTROY, oDocument
```

This code modifies the name node to change the airport to Portland, Oregon, and writes the modified XML to a new file. Please note that if you save to an existing file (e.g., using sample.xml instead of sample2.xml at the end of this example), the current XML data will replace the file entirely.

# Creating New Data

You can create an IDLffXMLDOMDocument object and start adding nodes to it without loading a file.

```
oDocument = OBJ_NEW('IDLffXMLDOMDocument')
oElement = oDocument->CreateElement('myElement')
oVoid = oDocument->AppendChild(oElement)
oDocument->Save, FILENAME='new.xml'
OBJ_DESTROY, oDocument
```

This code creates the following XML file:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<myElement/>
```

Note that <myElement/> is XML shorthand for <myElement></myElement>.

## Example

The following lines create an XML file containing the lines shown in "About the DOM Structure" on page 510:

```
oDocument = OBJ_NEW('IDLffXMLDOMDocument')
oPluginElement = oDocument->CreateElement('plugin')
oPluginElement->SetAttribute, 'type', 'tab-iframe'
oVoid = oDocument->AppendChild(oPluginElement)
```

```
oNameElement = oDocument->CreateElement('name')
oText = oDocument->CreateTextNode('Weather.com Radar Image [DEN]')
oVoid = oNameElement->AppendChild(oText)
oVoid = oPluginElement->AppendChild(oNameElement)
oDescriptionElement = oDocument->CreateElement('description')
oText = oDocument-> $
   CreateTextNode('600 mile Doppler radar image for DEN')
oVoid = oDescriptionElement->AppendChild(oText)
oVoid = oPluginElement->AppendChild(oDescriptionElement)
oVersionElement = oDocument->CreateElement('version')
oText = oDocument->CreateTextNode('1.0')
oVoid = oVersionElement->AppendChild(oText)
oVoid = oPluginElement->AppendChild(oVersionElement)
oTabElement = oDocument->CreateElement('tab')
oIconElement = oDocument->CreateElement('icon')
oText = oDocument->CreateTextNode('weather.gif')
oVoid = oIconElement->AppendChild(oText)
oVoid = oTabElement->AppendChild(oIconElement)
oTooltipElement = oDocument->CreateElement('tooltip')
oText = oDocument->CreateTextNode('DEN Doppler radar image')
oVoid = oTooltipElement->AppendChild(oText)
oVoid = oTabElement->AppendChild(oTooltipElement)
oVoid = oPluginElement->AppendChild(oTabElement)

; Write the output to a file in the TEMP directory.
; Use the PRETTY_PRINT keyword to generate nicely-formatted
; output.
tmpdir = GETENV('IDL_TMPDIR')
oDocument->Save, FILENAME=tmpdir+'sample.xml', /PRETTY_PRINT
OBJ_DESTROY, oDocument

; Optionally, you can open the sample.xml file in the
; IDL Workbench editor:
PUSHD, tmpdir
.EDIT sample.xml
POPD
```

## Destroying IDLffXMLDOM Objects

Suppose that you are done with the name node and want to look at the description.

```
OBJ_DESTROY, oName
oDesc = oNodeList->Item(1)
oDescText = oDesc->GetFirstChild()
PRINT, oDescText->GetNodeValue()
```

This code destroys the oName object and oNameText with it because it was created
by oName's GetFirstChild method. This automatic destruction cleans up all the

objects that you might have created from the `oName` node. You can then fetch the description element from the node list and print its name in the same manner. The name node is still in the node list and can be fetched again from the node list with the Item method, if needed.

Finally,

```
OBJ_DESTROY, oDocument
```

destroys the top-level object that you originally created with the OBJ_NEW function and also destroys any other objects that were created directly or indirectly from the `oDocument` object.

You can write the first code sample above more compactly because of the ability of the IDLffXMLDOMDocument object to clean up all the objects it and its children created:

```
oDocument = OBJ_NEW('IDLffXMLDOMDocument')
oDocument->Load, FILENAME='sample.xml'
PRINT, ((((oDocument->GetFirstChild())-> $
   GetElementsByTagName('name'))-> $
   Item(0))->GetFirstChild())->GetNodeValue()
OBJ_DESTROY, oDocument
```

Under normal circumstances, the three object references created by the calls to the GetFirstChild and GetElementsByTagName methods would be lost because the object references to these three objects were not stored in IDL user variables. However, these objects are cleaned up by the document object when it is destroyed.

For additional information, see "Orphan Nodes" on page 526.

Please note:

- In general, you should not use the OBJ_NEW function to create any IDLffXMLDOM objects except for the top-level document object. Use the methods such as GetFirstChild to create the objects.

- You can destroy objects obtained from the various methods (e.g., GetFirstChild) at any time by the OBJ_DESTROY procedure.

- Objects destroyed by the OBJ_DESTROY procedure also destroy objects that they created.

- Destroying objects does *not* modify the DOM structure. That is, destroying any of the IDLffXMLDOM objects does not modify the data in the DOM tree. There are explicit methods for modifying DOM tree data. Destroying IDLffXMLDOM objects only removes your ability to access the DOM tree data.

# Working with Whitespace

The XML parser is very particular about whitespace because all characters in an XML document define the content of that document. Whitespace consists of spaces, tabs, and newline characters, all of which are commonly used to format documents to make them easier to work with. In many cases, this whitespace is unimportant with respect to the document content. It is there only for presentation and does not affect the actual data stored in the XML document. However, in some cases, for example with CDATA or text node information, the whitespace might be important.

When whitespace is not important, IDL can treat it as ignorable. In many circumstances, you might want the parser to skip over this ignorable whitespace and not place it in the DOM tree so that you do not need to deal with it when visiting nodes in the DOM tree.

For example, the following two XML fragments produce different DOM trees when parsed with the default parser settings:

```
<stateList>
   <state>Colorado</state>
</stateList>

<stateList><state>Colorado</state></stateList>
```

In the first fragment, the stateList element has two child nodes that the second fragment does not. They are text nodes containing whitespace, a newline, and some tabs or spaces.

For the parser to distinguish between non-ignorable and ignorable whitespace, there must be a DTD associated with the XML document, and it must be used to validate the document during parsing. This implies that a VALIDATION_MODE of 1 or 2 must be used when loading the XML document with the IDLffXMLDOMDocument::Load method.

Once validation is established, you can either:

- Tell the parser not to include ignorable text nodes in the DOM tree by setting the EXCLUDE_IGNORABLE_WHITESPACE keyword in the IDLffXMLDOMDocument::Load method. If you select this option, the DOM trees for each of the above two fragments are the same.

- Check each text node in the DOM tree with the IDLffXMLDOMText::IsIgnorableWhitespace method.

# Orphan Nodes

You can remove nodes from the DOM tree by using the
IDLffXMLDOMNode::RemoveChild and IDLffXMLDOMNode::ReplaceChild
methods. When these nodes are removed from the tree, they are owned by the DOM
document directly and have no parent (since they are not in the tree anymore).
Similarly, when these methods are used, the IDLffXMLDOM objects' ownership is
changed as well because the IDL tree (made by creating the document interface and
adding nodes) must mirror the underlying DOM tree.

If you issue the following command:

```
oMyRemovedChild = oMyElement->RemoveChild(oMyChild)
```

oMyChild is no longer owned by oMyElement and becomes owned by the
document object to which all these nodes belong. Here, oMyRemovedChild and
oMyChild are actually object references to the same object. The function method
syntax provides a convenient way to create a new object reference variable with a
new name that reflects the new status of the removed object, and you can use either
name to access the orphaned node.

After removal, the orphan node is loosely associated with the document via the
ownership relationship and would not be included in the output if the DOM tree were
written to a file. You can insert the node back into the DOM tree with an InsertBefore
or AppendChild method.

If the document that contains orphan nodes is destroyed, the orphan nodes are lost.
More specifically, DOM tree orphan nodes are not written out to a file if they are
orphans at the time that the IDLffXMLDOMDocument::Save method is used to save
the tree, and the IDL node objects referring to the orphans are destroyed when the
document object is destroyed.

# Tree-Walking Example

The following code traverses a DOM tree using pre-order traversal.

```
PRO sample_recurse, oNode, indent

   ; "Visit" the node by printing its name and value
   PRINT, indent GT 0 ? STRJOIN(REPLICATE(' ', indent)) : '', $
      oNode->GetNodeName(), ':', oNode->GetNodeValue()

   ; Visit children
   oSibling = oNode->GetFirstChild()
   WHILE OBJ_VALID(oSibling) DO BEGIN
      SAMPLE_RECURSE, oSibling, indent+3
      oSibling = oSibling->GetNextSibling()
   ENDWHILE
END

PRO sample
   oDoc = OBJ_NEW('IDLffXMLDOMDocument')
   oDoc->Load, FILENAME="sample.xml"
   SAMPLE_RECURSE, oDoc, 0
   OBJ_DESTROY, oDoc
END
```

This program generates the following output for the plug-in file (see "About the DOM Structure" on page 510):

```
#document:
   plugin:
      #text:

      name:
         #text:Weather.com Radar Image [DEN]
      #text:

      description:
         #text:600 mile Doppler radar image for DEN
      #text:

      version:
         #text:1.0
      #text:

      tab:
         #text:

         icon:
```

```
          #text:weather.gif
      #text:

      tooltip:
          #text:DEN Doppler radar image
      #text:

  #text:
```

The program above created an IDLffXMLDOM object for every node it encountered and did not destroy them until the document was destroyed. Another approach, illustrated in the program below, cleans up the nodes as it proceeds:

```
PRO sample_recurse2, oNode, indent
   ;; "Visit" the node by printing its name and value
   PRINT, indent gt 0 ? STRJOIN(REPLICATE(' ', indent)) : '', $
      oNode->GetNodeName(), ':', oNode->GetNodeValue()

   ;; Visit children
   oNodeList = oNode->GetChildNodes()
   n = oNodeList->GetLength()
   for i=0, n-1 do $
      SAMPLE_RECURSE2, oNodeList->Item(i), indent+3
   OBJ_DESTROY, oNodeList
END

PRO sample2
   oDoc = OBJ_NEW('IDLffXMLDOMDocument')
   oDoc->Load, FILENAME="sample.xml"
   SAMPLE_RECURSE2, oDoc, 0
   OBJ_DESTROY, oDoc
END
```

Please note that document and text nodes do not have node names, so the GetNodeName method always returns '#document' and '#text,' respectively.
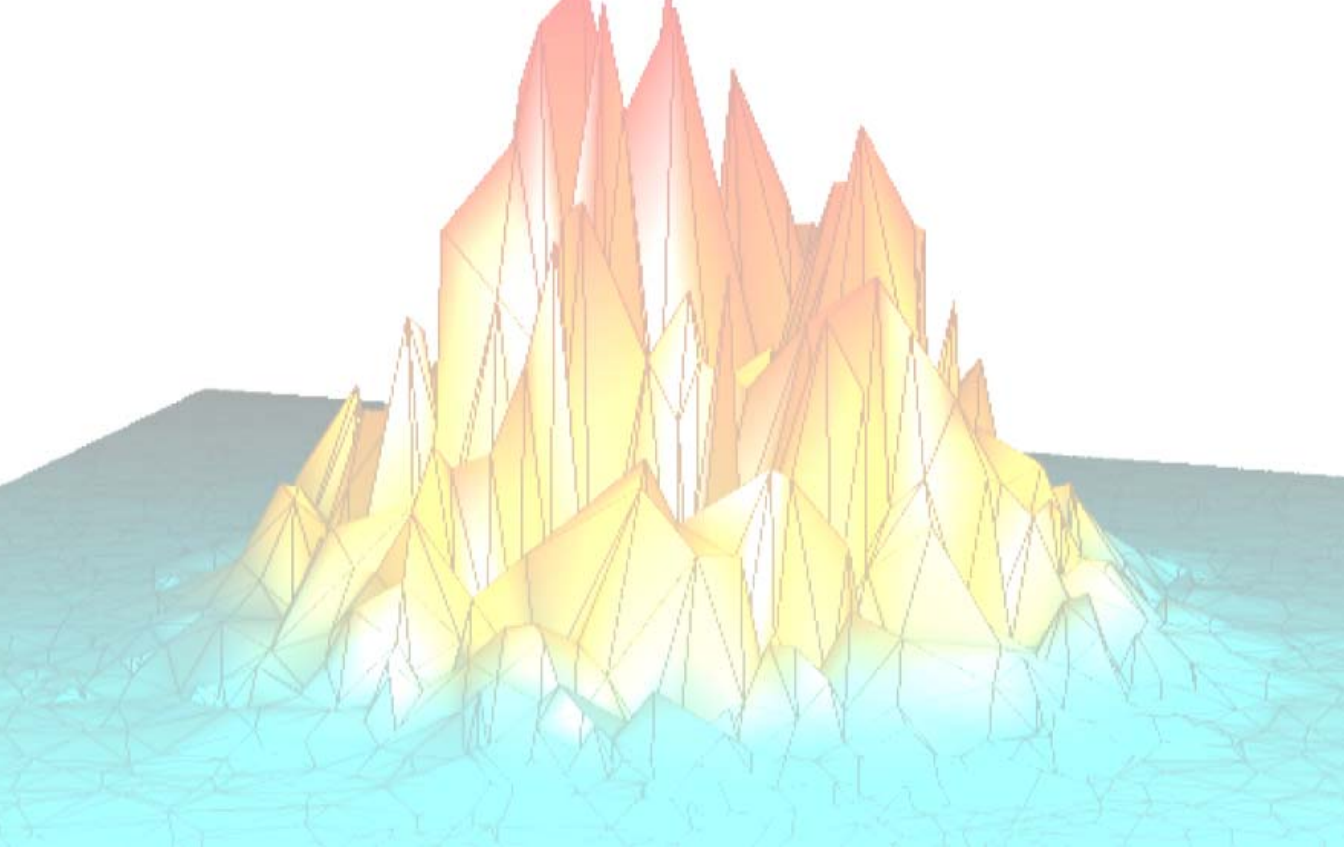
# Part III: Creating Applications in IDL

# Chapter 22

# Providing Online Help For Your Application

The following topics are covered in this chapter:

# Overview of Creating Application Help

IDL gives you the ability to display help information for your applications, routines, *etc.* using a variety of mechanisms:

- Using tooltips, status bars, and text widgets to display small amounts of help information within an application's interface.

- Using the XDISPLAYFILE procedure to display text files in an IDL window separate from your application.

- Using the SPAWN procedure to display a file in an external editor or viewer.

- Using IDL's own online help facilities, via the ONLINE_HELP procedure, to display Windows Help files, Adobe Portable Document Format files, or HTML files.

These techniques vary in complexity, cost, and level of integration with IDL and your own application. The following sections describe each option in detail.

# Providing Help Within the User Interface

There are numerous ways to supply help and feedback to users of a widget application without the need to display a help file in an external window. The following techniques can augment, if not necessarily replace, a more complete online help file.

## Tooltips

Tooltips are short text strings that appear when the mouse cursor is positioned over a button or draw widget for a few seconds. Often a tooltip is enough to remind a user of the function of a button, eliminating the need for the user to consult more extensive documentation.
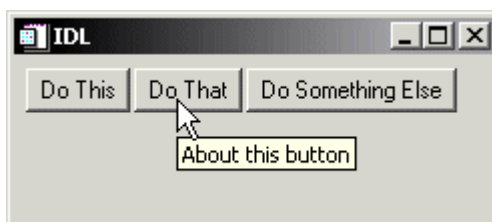


*Figure 22-1: A Tooltip*

Tooltips are created by specifying a text string as the value of the TOOLTIP keyword to the WIDGET_BUTTON function:

```
DoneButton = WIDGET_BUTTON(base, VALUE='Done', $
    TOOLTIP='Click here to close the application')
```

**Note** ———————————————————————————————————————————
  Draw widgets can also display tooltips.
————————————————————————————————————————————————————

## Status Lines

You can give users feedback about the status of an operation or the function of an interface element by updating a status line included in your widget interface. Status lines are generally located at the bottom of the interface, and can be updated as the

user moves the mouse cursor over interface elements or as the status of the application changes.
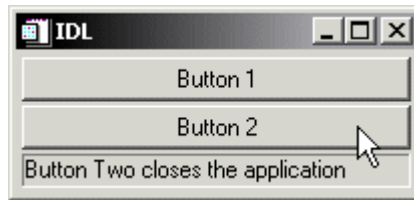


*Figure 22-2: A status line.*

The following example demonstrates how a status line can be updated as the mouse cursor moves over a set of buttons. Similar code could update the value of the label widget as other events occur. To view the results, paste the code into an IDL editor window and save it as label_update.pro, then compile and run.

```
; Event-handler routine
PRO label_update_event, ev

; If the event is a tracking event, update the label widget.
IF (TAG_NAMES(ev, /STRUCTURE) EQ 'WIDGET_TRACKING') THEN BEGIN
   WIDGET_CONTROL, ev.TOP, GET_UVALUE=label
   WIDGET_CONTROL, ev.ID, GET_VALUE=val, GET_UVALUE=uval
   WIDGET_CONTROL, label, SET_VALUE=uval
   WIDGET_CONTROL, label, SET_VALUE=uval
ENDIF

; If the event is a button event, and comes from Button 2,
; then destroy the application.
IF (TAG_NAMES(ev, /STRUCTURE) EQ 'WIDGET_BUTTON') THEN BEGIN
   WIDGET_CONTROL, ev.ID, GET_VALUE=val
   IF (val EQ 'Button 2') THEN WIDGET_CONTROL, ev.TOP, /DESTROY
ENDIF

END

; Widget creation routine
PRO label_update

base=WIDGET_BASE(/COLUMN, XSIZE=200)

; Set the button widgets to generate tracking events, so we
; know when the mouse cursor is over them.
b1 = WIDGET_BUTTON(base, VALUE='Button 1', $
```

```
    UVALUE='Button One does nothing', /TRACKING_EVENTS)
b2 = WIDGET_BUTTON(base, VALUE='Button 2', $
    UVALUE='Button Two closes the application', /TRACKING_EVENTS)
label = WIDGET_LABEL(base, XSIZE=190, /SUNKEN_FRAME)

; Set the user value of the base widget equal to the widget ID
; of the label widget.
WIDGET_CONTROL, base, SET_UVALUE=label

; Realise the widgets and call XMANAGER.
WIDGET_CONTROL, base, /REALIZE
XMANAGER, 'label_update', base

END
```

## Text Widgets

To display larger amounts of text than will fit conveniently in a status line, you can include a text widget in your application's interface. The process of updating the text widget's value depending on user actions is similar to the process described in the status line example, above.

To display larger blocks of text that would not fit conveniently within the body of your application's interface, consider using the XDISPLAYFILE procedure as described in "Displaying Text Files" on page 536.

# Displaying Text Files

The IDL XDISPLAYFILE procedure displays an ASCII text file using a predefined widget interface. To see an example, enter the following statement at the IDL command prompt:

```
XDISPLAYFILE, FILEPATH('relnotes.txt')
```

This command displays the current release notes file for your IDL installation in a widget interface.

To display your own text file, create a "Help" button of some sort in your widget interface and configure the button's event handling procedure to call XDISPLAYFILE with the full path to the text file.

See "XDISPLAYFILE" (*IDL Reference Guide*) for more details.

**Note** ─────────────────────────────────────────────

By default, the XDISPLAYFILE window exists separately from your application, and will not be closed when your application exits. To ensure that the XDISPLAYFILE window closes when your application exits, set the value of the GROUP keyword equal to the widget ID of your application's top-level base. See "Using Multiple Widget Hierarchies" (Chapter 3, *User Interface Programming*) for a discussion of widget grouping.

─────────────────────────────────────────────────────

# Using an External Viewer

If you are certain that a specific viewing application is present on the system on which your application will run, you can use the IDL SPAWN procedure to display a help file using that application.

Note that you must have some fairly explicit information about the system on which your application will run to use this technique. You must know:

-   that the application you wish to use is installed on the system, and

-   the full path to the application's executable file.

(If your application is complex enough to have an installation program or procedure, you might be able to query the user for the path to the external viewer at installation time.)

**Note**

If you want to display HTML or Portable Document Format (PDF) files, see "Using Other Online Help Viewers" on page 539.

For example, suppose you know that your application will run on a Windows system, you could open a text file in the Notepad application, which is always located in the Windows system directory and can be invoked without specifying a full path:

```
SPAWN, 'notepad.exe D:/myapp/myfile.txt', /NOSHELL, /NOWAIT
```

For more information, see "SPAWN" (*IDL Reference Guide*).

# About IDL's Online Help System

Beginning with IDL version 7.0, IDL's context-sensitive online help system is built on the user assistance infrastructure provided by the Eclipse framework on which the IDL Workbench is built. Help content for IDL is provided in a set of *help plugins* — .jar archives that contain HTML help content files and XML documents that control how the content is presented.

IDL's online help system is described in detail from a user's point of view in Using IDL Help — located, appropriately enough, in the IDL online help system.

In IDL 7.0, it is not possible for IDL developers to write help content to be displayed in IDL's own help system. We hope to provide this capability to IDL developers in a future release. In the meantime, see "Using Other Online Help Viewers" on page 539 for options on providing help for your IDL applications.

# Using Other Online Help Viewers

You can use the ONLINE_HELP procedure to display help files in several formats. The type of help file or files you choose to create will depend on the platforms on which your IDL application will be used, and on your own preferences.

- IDL Assistant Help Systems
- Microsoft Windows Help
- Portable Document Format Files
- HTML Files

## IDL Assistant Help Systems

IDL versions 6.2 through 6.4 used a cross-platform help viewer — *IDL Assistant* — based on the help viewer used by the Qt development toolkit from Trolltech. Although the IDL Assistant help viewer has been replaced as IDL's default help viewer in version 7.0, it is still included in IDL distributions as an option for user-created help systems.

The process of creating help systems for that use the IDL Assistant is somewhat complex. See "Using the IDL Assistant Help System" on page 545 for complete details.

## Microsoft Windows Help

There are currently two Windows online help formats in wide use: WinHelp and HTML Help. WinHelp is the older of the two, and many applications still provide help in this format, which can be distinguished by the file extension ".hlp". HTML Help is the newer format, and provides (among other things) the ability to include links to documents in various formats, both local and network-based. HTML Help files use the file extension ".chm". Viewers for both types of online help are included in all relatively current versions of Windows, and IDL's ONLINE_HELP procedure will invoke the correct viewer for either type of file.

### Creating Windows Help Files

Microsoft Windows help files are relatively easy to create. Files in a specified format (the Rich Text Format, (RTF) for WinHelp, or a wider variety of formats for HTML Help) are compiled with a help compiler from Microsoft. The help compiler is part of the Windows Software Developer's Kit, and is now included in several Microsoft programming products, including the Visual C++ development environment. The

help compiler may also be available from the Microsoft Web site or other Microsoft online software libraries at little or no cost.

It is beyond the scope of this manual to discuss the preparation and compilation of Windows help files. Microsoft provides useful information about its help-system products as part of the Microsoft Developer's Network; try searching the MSDN site at `http://msdn.microsoft.com` with the search term "HTML Help" or "WinHelp". There are also numerous third-party books on creating Windows help systems available.

## Calling Windows Help Files

To call a Windows help file of either type from within IDL, use the ONLINE_HELP procedure. Specify the name of your help file using the BOOK keyword, and optionally specify a search term in the *Value* argument. Alternatively, you can specify a context number in the *Value* argument and include the CONTEXT keyword. See "ONLINE_HELP" (*IDL Reference Guide*) for details.

Depending on where your application and its help files are installed, you may also need to specify the full path to the file and the FULL_PATH keyword.

### Example 1

Suppose you have created an HTML Help file named `myapp.chm` to accompany your IDL application. Use the following call to open the HTML Help viewer and load the search term "controls" into the Index dialog:

```
ONLINE_HELP, 'controls', BOOK='path\myapp.chm', /FULL_PATH
```

where *path* is the full path to the file `myapp.chm`.

### Example 2

Suppose you have created a WinHelp file named `myapp.hlp` and placed it in the `Help` subdirectory of your IDL installation. If you know that the context number of the topic you wish to display is 250, use the following call to open the WinHelp viewer to the correct topic:

```
ONLINE_HELP, 250, BOOK='myapp', /CONTEXT
```

If no file extension is included in the value of the BOOK keyword, IDL will search each directory in !HELP_PATH until it finds a matching file with one of the following file extensions, in this order: `.chm` (Windows only), `.hlp` (Windows only), `.pdf`, `.html`, `.htm`. See "Paths for Help Files" on page 561 for details on setting the help path.

### Cross-Platform Issues

Windows help files (of either format) are viewable only on Microsoft Windows platforms. If your IDL application will be available on UNIX platforms as well as Microsoft Windows platforms, you have several options:

- Create help content suitable for use by the cross-platform IDL Assistant help viewer. See "About IDL's Online Help System" on page 538 for details on creating help content that will display in IDL Assistant.

- Create separate help files (one in Windows Help format, one in PDF or HTML format) and issue the appropriate call to ONLINE_HELP based on the current platform. If you name the files with the same base name (but with different file extensions), IDL will automatically select the correct file for the platform.

- Create a single help file in PDF or HTML format, and caution your users that they must have a the appropriate viewing application installed in order to use your help file. In addition, UNIX users must ensure that the viewing application is properly configured for use by IDL, as described in "Displaying HTML and PDF Files under UNIX" under "ONLINE_HELP" (*IDL Reference Guide*).

## Portable Document Format Files

You can use the ONLINE_HELP procedure to display a PDF file on any system that has a PDF-display application installed.

**Note** ───────────────────────────────────────────────

IDL launches a stand-alone version of the PDF viewing application. Files are *not* displayed in the Windows help viewer or any other browser application.

─────────────────────────────────────────────────────

### Creating PDF Files

To create PDF files for use with IDL's online help system, you will need an application that allows you to author PDF files or convert files in other formats to PDF. Most commonly, source files are created with a text-editor, word-processor, or other document-production program, printed to a PostScript file, and run through a program that *distills* the PostScript into PDF. Adobe's commercial Acrobat package includes the Acrobat Distiller, which provides a convenient GUI interface to the distillation process. Other third-party software to distill PostScript files into PDF is also available; GhostScript (`www.ghostscript.com`) is one freely available alternative.

It is beyond the scope of this manual to discuss creation of PDF files in detail; consult the documentation for your PDF authoring system or distilling software for details.

## Calling PDF Files

To call a PDF help file from within IDL, use the ONLINE_HELP procedure. Specify the name of your PDF file using the BOOK keyword. Depending on where your application and its help files are installed, you may also need to specify the full path to the file and the FULL_PATH keyword.

See "ONLINE_HELP" (*IDL Reference Guide*) for details.

### Example 1

Suppose you have created a PDF file named myapp.pdf to accompany your IDL application. Use the following call to open the PDF viewer and display the first page of the file:

```
ONLINE_HELP, BOOK='path\myapp.pdf', /FULL_PATH
```

where *path* is the full path to the file myapp.pdf.

### Example 2

If the myapp.pdf file is located in one of the directories included in IDL's !HELP_PATH system variable, you do not need to include either the .pdf extension or the FULL_PATH keyword:

```
ONLINE_HELP, BOOK='myapp'
```

If no file extension is included in the value of the BOOK keyword, IDL will search each directory in !HELP_PATH until it finds a matching file with one of the following file extensions, in this order: .chm (Windows only), .hlp (Windows only), .pdf, .html, .htm. See "Paths for Help Files" on page 561 for details on setting the help path.

## Cross-Platform Issues

If you intend to use PDF files to supply online help for your cross-platform application, you should caution your users that they must have a the appropriate PDF viewing application installed in order to use your help file. In addition, UNIX users must ensure that the viewing application is properly configured for use by IDL, as described in "Displaying HTML and PDF Files under UNIX" under "ONLINE_HELP" (*IDL Reference Guide*).

# HTML Files

You can use the ONLINE_HELP procedure to display an HTML file on any system that has a Web-browser installed. On UNIX systems, the browser's executable file must also be in a directory included in the PATH environment variable.

## Creating HTML Files

It is beyond the scope of this manual to discuss HTML authoring in detail. Use any technique you are comfortable with to create HTML files for display in a normal Web browser.

**Note**
You can use the MK_HTML_HELP procedure to create HTML-formatted documentation for your application from standard IDL documentation headers. See "MK_HTML_HELP" (*IDL Reference Guide*) for details.

## Calling HTML Files

To call an HTML file from within IDL, use the ONLINE_HELP procedure. Specify the name of your HTML file using the BOOK keyword. Depending on where your application and its help files are installed, you may also need to specify the full path to the file and the FULL_PATH keyword.

See "ONLINE_HELP" (*IDL Reference Guide*) for details.

### Example 1

Suppose you have created an HTML file named myapp.html to accompany your IDL application. Use the following call to open the default Web browser and display the file, positioned to the HTML anchor tag anchor1:

```
ONLINE_HELP, 'anchor1', BOOK='path\myapp.html', /FULL_PATH
```

where *path* is the full path to the file myapp.html.

### Example 2

If the myapp.html file is located in one of the directories included in IDL's !HELP_PATH system variable, you do not need to include the .html extension or the FULL_PATH keyword:

```
ONLINE_HELP, BOOK='myapp'
```

If no file extension is included in the value of the BOOK keyword, IDL will search each directory in !HELP_PATH until it finds a matching file with one of the

following file extensions, in this order: .chm (Windows only), .hlp (Windows only), .pdf, .html, .htm. See "Paths for Help Files" on page 561 for details on setting the help path.

## Cross-Platform Issues

If you intend to use HTML files to supply online help for your cross-platform application, keep the following things in mind:

- IDL does not require that a Web browser be installed. While it is unlikely that you will encounter systems that do not include a Web browser, you may wish to inform your users in advance that your application uses a Web browser to supply help.

- On UNIX systems, it may be necessary to modify IDL's default HTML browser configuration script to use a locally-preferred browser. See "Displaying HTML and PDF Files under UNIX" under "ONLINE_HELP" (*IDL Reference Guide*) for details.

- Different browsers contain different display engines, and may display HTML in different ways. This is especially true if you use features that have only recently been added to the HTML specification. Check for display issues using as many browsers as you reasonably can.

# Using the IDL Assistant Help System

IDL versions 6.2 through 6.4 used a cross-platform help viewer — *IDL Assistant* — based on the help viewer used by the Qt development toolkit from Trolltech. Although the IDL Assistant help viewer has been replaced as IDL's default help viewer in version 7.0, it is still included in IDL distributions as an option for user-created help systems.

This section discusses the following topics relating to creating help systems for the IDL Assistant help viewer:

- "Using the IDL Assistant Help Viewer" on page 545
- "Format of an IDL Assistant Help System" on page 552
- "Creating Help Content" on page 552
- "Creating an Assistant Document Profile" on page 553
- "Optional Help System Files" on page 559
- "Displaying Help Topics" on page 560
- "Paths for Help Files" on page 561

## Using the IDL Assistant Help Viewer

This section describes how to use the IDL Assistant application. For information on creating help content that uses the IDL Assistant for your own IDL applications, see the sections that follow.

### The Main Window

The IDL Assistant *main window* contains the text of the current topic. Within the main window you can:

- Follow hypertext links to other topics, or to sections within the current topic
- Navigate to the next or preceding topic using arrows at the top of the topic screen
- Display multiple topics simultaneously using the tabbed interface
- Create new tabs and close existing tabs using icons to the right and left of the tabs

- Perform common tasks including display of the next/previous topic, tab management, text sizing, copying text to the clipboard, and finding text within the topic using the context menu

## The Sidebar

The IDL Assistant *sidebar* provides four tabs that allow you to navigate through the specified documentation set. All of the tabs provide a context menu that allows you to open the selected topic the current tab, a new tab, or a new window.

### The Contents Tab

The **Contents** tab displays a hierarchical listing of the contents of the various books in the specified documentation set.

### The Index Tab

The **Index** tab provides a keyword index of the contents of the specified documentation set. Enter a text string in the **Look For:** field to see keywords that match the string.

### The Search Tab

The **Search** tab allows you to search the text of the specified documentation set for words or phrases. Text matching your search string is highlighted when a topic is displayed in the main window.

**Tip**
Words or phrases entered in the **Search** tab are *not* case sensitive.

To search for words, enter one or more strings in the **Searching for:** field, separated by spaces and click **Search**. IDL Assistant displays a list of topics that contain all of the words you entered.

To search for a phrase, enclose the phrase in single or double quote marks.

The list of topics containing the search words or phrase is displayed as a list ranked roughly according to the number of occurrences of the words or phrases, with the topics containing the largest number of occurrences listed given higher rankings.

**Allowed Characters**

The following characters are allowed in the **Search** tab:

- Letters (upper- and lower-case)
- Numbers (0-9)

- Quote marks (single ( ' ), double ( " ), backwards ( ` )  )

- Exclamation marks ( ! ), colons ( : ), and periods ( . )

- Spaces

- Hyphens ( - )

- Underscores ( _ )

- Asterisk ( * ) as a wildcard matching one or more unspecified characters

    **Note**
    The * character cannot be used within quotes or at the beginning of a string.

All other characters are disallowed; you cannot enter them in the **Searching for:** field.

**Warning**
Searches that contain single-character strings (such as "a" or "8") are not allowed and will return no results. This is true even when the single character is combined with a punctuation character such as a hyphen. For example, searching for the string "8-bit" will return no results.

**Examples**

| | |
|---|---|
| convol | List all topics that contain the word "convol" |
| convol* | List all topics that contain a word *beginning* with "convol" |
| base widget | List all topics that contain the word "base" *and* the word "widget" |
| "base widget" | List all topics that contain the phrase "base widget" |

**The Bookmarks Tab**

The **Bookmarks** tab allows you to save links to specific topics in the IDL documentation set for easy reference.

## The Menu Bar

The IDL Assistant *menu bar* runs across the top of the IDL Assistant window, and provides access to the features listed below. Keyboard shortcuts to invoke various menu items are listed in the menus themselves.

| Menu | Item | Function |
|------|------|----------|
| File | **New Window** | Open a new IDL Assistant window. |
|      | **Add Tab** | Open a new tab displaying the same topic as the currently selected tab. |
|      | **Close Tab** | Close the currently selected tab. |
|      | **Print** | Print the contents of the currently selected tab. See "Printing" on page 551 for details. |
|      | **Close** | Close the current IDL Assistant window. |
|      | **Exit** | Close all IDL Assistant windows. |
| Edit | **Copy** | Copy text selected in the main window to the system clipboard. |
|      | **Find in Text...** | Search for a text string in the currently displayed topic. |
|      | **Find Next** | Find the next instance of the text string in the currently displayed topic. |
|      | **Find Previous** | Find the previous instance of the text string in the currently displayed topic. |
|      | **Settings...** | Display the Settings dialog. See "Settings" on page 551 for details. |

*Table 22-1: IDL Assistant Menus*

| Menu | Item | Function |
|------|------|----------|
| **View** | **Zoom in** | Increase the text size in the main window. See "Text Zoom" on page 550 for important notes. |
| | **Zoom out** | Decrease the text size in the main window. See "Text Zoom" on page 550 for important notes. |
| | **Views...** | Control display of the Sidebar and Standard toolbar.<br><br>**Note -** The Line Up feature realigns the toolbar if it has been moved. |
| **Go** | **Previous** | Display the current tab's previous topic. |
| | **Next** | Display the current tab's next topic. |
| | **Home** | Display the IDL online help Home page. |
| | **Next Tab** | Select the tab to the right of the current tab, if any. |
| | **Previous Tab** | Select the tab to the left of the current tab, if any. |
| **Bookmark** | **Add Bookmark** | Create a bookmark for the currently selected topic. |
| | *Bookmark list* | Existing bookmarks are displayed at the bottom of this menu. |
| **Help** | **IDL Assistant Manual** | Display this help topic. |
| | **About IDL Assistant** | Display information about IDL Assistant. |
| | **What's This?** | Display context-sensitive pop-up help about some portions of the IDL Assistant interface. |

*Table 22-1: IDL Assistant Menus*

## The Tool Bar

The IDL Assistant *tool bar* provides quick access to a subset of the features available via the menubar.

| Icon | Name | Function |
|------|------|----------|
| ⬅ | Previous | Display the current tab's previous topic. |
| ➡ | Next | Display the current tab's next topic. |
| 🏠 | Home | Display the IDL online help Home page. |
| 📄 | Copy | Copy text selected in the main window to the system clipboard. |
| 🔍 | Find in Text | Search for a text string in the currently displayed topic. |
| 🖶 | Print | Print the contents of the currently selected tab. See "Printing" on page 551 for details. |
| 🔍⁺ | Zoom in | Increase the text size in the main window. See "Text Zoom" on page 550 for important notes. |
| 🔍⁻ | Zoom out | Decrease the text size in the main window. See "Text Zoom" on page 550 for important notes. |
| ⬉? | What's this? | Display context-sensitive pop-up help about some portions of the IDL Assistant interface. |

*Table 22-2: IDL Assistant Toolbar*

## Text Zoom

Select **Zoom in** or **Zoom out** from the **View** menu to change the size of the text in the IDL Assistant main window.

The smoothness of the text zoom operation depends on the ability of the operating system to provide fonts of the appropriate size for the zoomed text. On platforms that provide robust font-management mechanisms, the **Zoom** operations will work smoothly. On platforms that provide more limited font support, a single **Zoom** operation may, depending on the current text size and font support, change the text size for only some text elements in the main window, or none at all. In these cases, repeated applications of the **Zoom** operations may change the text size.

If you find that the text zooming feature does not work adequately with the default fonts, try changing the fonts used by IDL Assistant (see "Settings" on page 551 for details.) On platforms that use a set of fixed-size fonts, choosing a font with a larger number of available sizes will allow smoother text zooming.

## Printing

Select **Print** from the **File** menu or toolbar to display a platform-native **Print** dialog that allows you to select a printer on which to print.

**Note**

Currently, the only text range option available is **All**. Printing all will print the entire contents of the topic currently displayed in the main window.

**Tip**

The quality of the printed output from IDL Assistant depends on the platform and printer in use. For high-quality printed output, consider printing from the PDF version of the document you are viewing.

## Settings

Select **Settings** from the **Edit** menu to display a tabbed dialog that allows you to control several IDL Assistant settings.

### General Tab

The **General** tab allows you to select fonts for text display in the main window. By default, the **Font** is set to Helvetica, and the **Fixed Font** is set to Courier.

**Tip**

Depending on the configuration of your system, you may be able to select alternate fonts that provide better appearance or smoother zooming behavior than the defaults. This is especially true on UNIX systems that have a limited set of fonts available. Trying different font settings may improve both the legibility of the text and the ability to zoom in the IDL Assistant viewer.

The **General** tab also allows you to select a color for hyperlinks and specify whether the links should be underlined. Depending on your platform, changing these values may not produce the effect you expect.

### Web Tab

The **Web** tab allows you to define the web browser that should be invoked when you click on a hyperlink that refers to a web site rather than to a file in the IDL documentation set.

The **Web** tab also allows you to specify an HTML file that should be displayed when you select **Home** from the **Go** menu or click the **Home** toolbar icon.

### PDF Tab

The **PDF** tab allows you to define a Portable Document Format (Adobe Acrobat) file browser that should be invoked when you click on a hyperlink that refers to a PDF file.

**Note** ───────────────────────────────────────────────────────────

If you choose to define your PDF file browser as Adobe Acrobat, you must use version 7 or later.

───────────────────────────────────────────────────────────────────

# Format of an IDL Assistant Help System

The IDL Assistant help viewer displays basic HTML-format files that use a subset of the tags defined by the HTML 3.2 specification. The help viewer does not handle Cascading Style Sheets, Javascript, or frames. Basic HTML tables are supported, but some table features defined in HTML 3.2 — notably the <CAPTION> tag and explicit control of table column widths — are not supported.

An IDL Assistant help system consists of:

- HTML content files and image files that are referenced by the HTML files via the <IMG> tag. See "Creating Help Content" below.

- An Assistant Document Profile (.adp) file that defines both the hierarchical structure of the documentation (the table of contents) and its keyword index. See "Creating an Assistant Document Profile" on page 553.

- Several optional files, described in "Optional Help System Files" on page 559.

# Creating Help Content

You can create HTML-format help content using any text editor or HTML authoring tool. Make sure that HTML files you intend to display in the IDL Assistant help viewer do not incorporate Javascript, JScript, ActiveX elements, or frames.

### HTML Formatting

You can use most of the text formatting tags supported by the HTML 3.2 format in files intended for display in the IDL Assistant help viewer. If you include Cascading Style Sheet information, it will be quietly ignored by the help viewer.

### Directory Structure and File Naming

Place all of the HTML content files for your help system in the same directory that contains your .adp file. You are free to choose any file naming convention you prefer for your help system's HTML files. Note, however, that IDL will interpret the *Value* argument to the ONLINE_HELP procedure as the name of an HTML file in the same directory as your .adp file. See "ONLINE_HELP" (*IDL Reference Guide*) for additional details related to how IDL interprets the *Value* argument.

### Image Files

Image files referenced by your help system's HTML files can be in PNG, GIF, or JPEG format. Image files do not need to be in the same directory as the HTML content files for your help system; by convention, image files are stored in a subdirectory of the content directory.

## Creating an Assistant Document Profile

The .adp file is an XML-format file that defines properties of your help system, constructs a hierarchical table of contents, and provides keyword index terms for your help topics.

You must ensure that your help system's .adp file is a valid XML file. This means that each element must contain values for all required attributes and must be properly closed. If the structure of the .adp file is not valid, IDL Assistant will fail to load the information in the .adp file, and no table of contents or index will be available for your help system.

The following is a very simple example of an .adp file that defines the help system properties and a single help topic with two keyword index terms:

```
<!DOCTYPE DCF>
<assistantconfig version="3.3.0">
   <profile>
      <property name="name">MyApp Version 1.2</property>
      <property name="title">My Help System</property>
      <property name="startpage">home.html</property>
      <property name="aboutmenutext">About My App</property>
      <property name="abouturl">about_my_app.txt</property>
```

```
         <property name="assistantdocs">.</property>
      </profile>
      <DCF ref="my_home.html" title="My Help">
         <section ref="Topic1.html" title="Topic1">
            <keyword ref="Topic1.html">Index one</keyword>
            <keyword ref="Topic1.html#anchor">Index two</keyword>
         </section>
      </DCF>
   </assistantconfig>
```

The individual XML elements that make up an `.adp` file are described below.

## **<!DOCTYPE> Element**

The `.adp` file must begin with an XML `<!DOCTYPE>` element that defines the file as being of type "DCF." The first line of an `.adp` file must always be:

```
<!DOCTYPE DCF>
```

### **Element Value**

Elements of this type do not contain an element value, and do not need to be closed.

## **<assistantconfig> Element**

All of the content of the `.adp` file is enclosed in an `<assistantconfig>` element.

### **Element Value**

Elements of this type contain `<profile>` and `<DCF>` elements.

### **version Attribute**

When creating content for the IDL Assistant help viewer, set the `version` attribute to the value "3.3.0":

```
<assistantconfig version="3.3.0">
```

## **<profile> Element**

The `<profile>` element contains a set of `<property>` elements that define values used by the entire help system. The allowed attribute values are described in the <property> Element section, below.

### **Element Value**

Elements of this type contain `<property>` elements.

## **<property> Element**

Each <property> element defines a value used to configure the help viewer application.

### **Element Value**

The element value is a text string. Each element must include a name attribute with one of the attribute values listed below.

### **name**

The value of a <property> element with the name attribute set equal to "name" is the identifier for the help system. IDL Assistant will use this value when creating index, full-text search, and bookmark filenames for your help system. For example, the following <property> element defines the name of the help system as "MyApp Version 1.2":

```
<property name="name">MyApp Version 1.2</property>
```

### **title**

The value of a <property> element with the name attribute set equal to "title" is string displayed in the title bar of the IDL Assistant help viewer application window. For example, the following <property> element defines the title as "My Help System":

```
<property name="title">My Help System</property>
```

### **startpage**

The value of a <property> element with the name attribute set equal to "startpage" is a URL (relative to the .adp file) to the HTML file that will be displayed when a user clicks the IDL Assistant **Home** button or selects **Home** from the **Go** menu. For example, the following <property> element defines the start page as "home.html":

```
<property name="startpage">home.html</property>
```

**Note** ────────────────────────────────────────

When the ONLINE_HELP procedure opens a help system, if no HTML file is specified for display via the *Value* argument, the help viewer will attempt to open a file named home.html in the same directory as the .adp file. As a result, in most cases the value of the <property> element with the name attribute set equal to "startpage" should be home.html.

─────────────────────────────────────────────────

### aboutmenutext

The value of a `<property>` element with the `name` attribute set equal to "aboutmenutext" defines a string that will be included as a menu item in the IDL Assistant **Help** menu. Selecting the menu item displays the contents of the file defined by a `<property>` element with the `name` attribute set equal to "abouturl" in a modal dialog. For example, the following `<property>` element defines the Help menu item string as "About My App":

```
<property name="aboutmenutext">About My App</property>
```

This element is optional. If no `<property>` element with the `name` attribute set equal to "aboutmenutext" exists, the menu item is not displayed in the IDL Assistant **Help** menu.

### abouturl

The value of a `<property>` element with the `name` attribute set equal to "abouturl" is a URL (relative to the `.adp` file) to a text or HTML file that will be displayed in a modal dialog when the user selects the menu item defined by a `<property>` element with the `name` attribute set equal to "aboutmenutext". For example, the following `<property>` element defines the "About My App" menu item URL as "about_my_app.txt":

```
<property name="abouturl">about_my_app.txt</property>
```

This element is optional. If no `<property>` element with the `name` attribute set equal to "aboutmenutext" exists, there is no need to define this element.

**Warning**
> The "about" dialog is intended to display a small block of text. Some basic HTML text formatting is allowed, including font face, style, and point size. There is no explicit control over the size or configuration of the dialog.

### assistantdocs

The value of a `<property>` element with the `name` attribute set equal to "assistantdocs" is the path to the directory that contains the file `assistant.html`, which contains information on the use of the IDL Assistant help viewer. The path can be either absolute or relative to the directory that contains the `.adp` file. This file is displayed when the user selects **IDL Assistant Manual** from the **Help** menu.

The `assistant.html` file used by IDL Assistant itself is located in the `help/online_help` subdirectory of the IDL distribution. If you know the relative path from your `.adp` file to this location, you can include it in the `<property>` element and users of your help system will be able to display the "help on help"

content from the IDL online help system. If you do not know the relative path (perhaps because you do not know where users of your application will install it), you may wish to create your own `assistant.html` file containing "help on help" information.

**Note** ———————————————————————————————————————————

The file must be named `assistant.html`. The `<property>` element contains only the path to the directory that contains this file.

———————————————————————————————————————————

For example, suppose you know that your application (along with its help system) will only be installed on UNIX systems that have IDL installed in the default location (`/usr/local/itt`). You could set the value of the `<property>` element as follows to allow your users to view the "help on help" topic from the IDL online help system:

```
<property name="assistantdocs">
    /usr/local/itt/idl/help/online_help
</property>
```

Similarly, if you choose to create your own `assistant.html` file and place it alongside your other help system content, you could set the value of the `<property>` element as follows:

```
<property name="assistantdocs">.</property>
```

## **<DCF> Element**

A `<DCF>` element represents a single "book" in the help system, and encloses all of the `<section>` elements that make up the book. In the IDL Assistant help viewer, a `<DCF>` element is represented by a collapsible book icon in the **Contents** tab. Clicking on the book icon displays the topic associated with the `<DCF>` element in the main help window and either displays or collapses the hierarchy contained within the element in the **Contents** tab.

### **Element Value**

Elements of this type contain `<section>` elements.

### **ref Attribute**

The `ref` attribute of a `<DCF>` element specifies the path to the HTML file that will be displayed in the main window when the user clicks on the book icon in the **Contents** tab.

The path to the HTML file should be relative to the `.adp` file. You can optionally include an HTML anchor tag after the file name.

**title Attribute**

The title attribute of a `<DCF>` element specifies the text that will be displayed next to the book icon for the element in the **Contents** tab.

For example, the following `<DCF>` element specifies that the book icon for the enclosed group of topics will be titled "What's New" and will display the file whatsnew.html positioned to the HTML anchor tag anchor1:

```
<DCF ref="./whatsnew.html#anchor1" title="What's New">
```

## **<section> Element**

A `<section>` element represents a single topic in the help system. Topic titles are displayed in the table of contents. `<section>` elements can be nested; the hierarchy defined by the nested section elements is reflected in the Table of Contents display.

Clicking on the section title displays the topic associated with the `<section>` element in the main help window and either displays or collapses the hierarchy contained within the element in the **Contents** tab.

### **Element Value**

Elements of this type contain `<section>` and `<keyword>` elements.

### **ref Attribute**

The ref attribute of a `<section>` element specifies the path to the HTML file that will be displayed in the main window when the user clicks on the topic title in the **Contents** tab.

The path to the HTML file should be relative to the .adp file. You can optionally include an HTML anchor tag after the file name.

### **title Attribute**

The title attribute of a `<section>` element specifies the text that will be displayed as the section title in the **Contents** tab.

For example, the following nested `<section>` elements define three topics "contained" by the topic titled "Chapter 1":

```
<section ref="chap1.html" title="Chapter 1">
   <section ref="chap1a.html#anchor1" title="Subhead 1"></section>
   <section ref="chap1b.html#anchor1" title="Subhead 2"></section>
   <section ref="chap1b.html#anchor2" title="Subhead 3"></section>
</section>
```

### **<keyword> Element**

A <keyword> element defines an entry in the help system's keyword index. Keyword index entries are displayed in the **Index** tab.

#### **Element Value**

The element value is a text string that contains the keyword index entry text.

The keyword index may be hierarchical. If a <keyword> element's value string includes the colon character, the text will be treated as a multi-level index entry. Thus, the value

```
top level entry:subentry1
```

would be displayed in the **Index** tab as

```
top level entry
    subentry1
```

When the <keyword> element values are displayed in the Index tab, they are alphabetized by level. All of the top-level entries are alphabetized, and each top-level entry's subentries are then alphabetized.

#### **ref Attribute**

The ref attribute of a <keyword> element specifies the path to the HTML file that will be displayed in the main window when the user clicks on the entry in the **Index** tab.

The path to the HTML file should be relative to the .adp file. You can optionally include an HTML anchor tag after the file name.

For example, the following <keyword> element defines an index entry with the title "Thingamajig" that corresponds to the HTML anchor thingamajig in the HTML file myroutines1.html:

```
<keyword ref="myroutines1.html#thingamajig">Thingamajig</keyword>
```

# **Optional Help System Files**

The files described in this section are not required for your help system to function, but may be useful.

## **About file**

The "about" file is displayed when the user chooses the "about" entry from the IDL Assistant **Help** menu, if it exists. If you choose to create this file, it can be either

a text file or an HTML file containing basic HTML tags. See "aboutmenutext" and "abouturl" under "<property> Element" on page 555 for details.

### TopicNotFound.html

The `TopicNotFound.html` file is displayed when the *Value* argument to the ONLINE_HELP procedure is supplied, but the specified file is not found. See "Displaying Help Topics" below for additional information.

# Displaying Help Topics

To display a topic within your help system, use the ONLINE_HELP procedure, specifying name of your `.adp` file as the value of the BOOK keyword. For example, if your `.adp` file is named `myapp.adp`, and you have placed the help system in a directory that is included in IDL's help path, you would use the following ONLINE_HELP command:

```
ONLINE_HELP, BOOK="myapp.adp"
```

See "Paths for Help Files" on page 561 for more on setting IDL's help path. Alternatively, if you know the full path to the `.adp` file, you could use an ONLINE_HELP command like the following:

```
ONLINE_HELP, BOOK="/usr/local/myapp/help/myapp.adp", /FULL_PATH
```

In most cases, it is more appropriate to set IDL's help path to include your help system when your application runs, as described in "Adding a Directory to the Help Path at Runtime" on page 562.

To display a specific topic from your help system, include the *Value* argument to the ONLINE_HELP procedure:

```
ONLINE_HELP, "InterestingTopic", BOOK="myapp.adp"
```

When IDL executes this command, it will do the following things:

1. Attempt to locate the `myapp.adp` file in a directory contained in IDL's help path. If it cannot locate the `.adp` file, ONLINE_HELP exits with an error.

2. Look in the directory that contains `myapp.adp` for a file named INTERESTINGTOPIC with the extension `.html` or `.HTML`. If IDL finds this file, it is displayed in the help viewer's main pane, and the search ends.

3. Look in the directory that contains `myapp.adp` for a file named InterestingTopic with the extension `.html`. If IDL finds this file, it is displayed in the help viewer's main pane, and the search ends.

4. If neither version of the file specified by the *Value* argument is found, IDL attempts to display a topic named `TopicNotFound.html` in the help viewer's main pane. This file explains to the user that there is no file that matches the *Value* argument.

In general, your end-users should never see the `TopicNotFound.html` file, because you have control over the strings placed in the *Value* argument to the ONLINE_HELP procedure. If the possiblity that your end-users might supply a *Value* argument for a file that does not exist in your help system, create a `TopicNotFound.html` file and include it with your help system.

# Paths for Help Files

You can specify the search path for help files via the !HELP_PATH system variable. Placing your help files in a directory included in the help path means that you do not need to include the full path in your call to the ONLINE_HELP procedure; supplying the name of the help file is enough.

**Note** ───────────────────────────────────────────────────────────

IDL searches the directories specified by !HELP_PATH and chooses the first instance of a file that matches the name you specify via the BOOK keyword to ONLINE_HELP. If no file extension is included in the value of the BOOK keyword, IDL will search each directory in !HELP_PATH until it finds a matching file with one of the following file extensions, in this order: `.adp`, `.chm` (Windows only), `.hlp` (Windows only), `.pdf`, `.html`, `.htm`. You can override this behavior by explicitly specifying the desired file extension.

─────────────────────────────────────────────────────────────────────

By default, !HELP_PATH contains the `help` subdirectory of the main IDL directory. To change the default value of !HELP_PATH, change the value of the IDL_HELP_PATH preference.

To change the value of !HELP_PATH during a single IDL session, simply assign a new value to the system variable. For example, to add a directory of your choice to the end of the default help path, you could use the following command:

```
!HELP_PATH=!HELP_PATH+'mypath'
```

where *mypath* is a valid path string, including the appropriate path element separator character for your platform.

## Adding a Directory to the Help Path at Runtime

If you distribute your application to users who install it on their own systems, you have no way of knowing in advance how to set the value of !HELP_PATH.

Suppose you have an application named myapp, installed in an unknown location your end-user's computer. The help system for myapp is located in a subdirectory of your application's directory named help. Including the following block of code in myapp.pro would be one way to determine the location of your help system at runtime, and set the !HELP_PATH system variable accordingly.

```
myapp_info = ROUTINE_INFO('myapp', /SOURCE)
myapp_path = FILE_DIRNAME(myapp_info.path)
myapp_help_path = myapp_path + PATH_SEP() + 'help'
!HELP_PATH = !HELP_PATH + PATH_SEP(/SEARCH_PATH) + myapp_help_path
```

Once the help path is set in this manner, you can simply provide the name of the .adp file for your help system as the value of the BOOK keyword to the ONLINE_HELP procedure.

# Chapter 23
# Distributing Runtime Mode Applications

This chapter describes the process of creating IDL runtime applications for distribution.

# What Is an IDL Runtime Mode Application?

An *IDL runtime mode application* is a program or set of programs written to use IDL's data analysis and display capabilities in a stand-alone mode, without access to the IDL Workbench, the IDL command line, or the ability to compile IDL .pro files. All IDL code for a runtime mode application must be pre-compiled and provided in the binary SAVE file format. If a runtime mode application presents a user interface, it must be exposed via the IDL widget toolkit or iTools functionality, since no access to the IDL command line or command output log is provided to the user.

Runtime mode applications are generally intended for users who do not have an IDL development license, although users who *do* have a development license can execute runtime mode applications as well. Typically, a runtime mode application is distributed along with an IDL distribution hierarchy containing all of the files necessary to run the application. (The exception is an application written to be run in the IDL Virtual Machine, which is installed separately from the IDL application itself.)

**Note** ───────────────────────────────────────────────────────────────────

IDL applications written to run with an IDL development license — one that allows the application to compile .pro files and access to the IDL command line — can, of course, be distributed to other IDL users. Distributing applications that run with an IDL development license can be as simple as providing the application files to the end user along with instructions describing how to install the files and configure the application.

───────────────────────────────────────────────────────────────────────────

This chapter describes the process of packaging an application written entirely in IDL so that it can be distributed to end users who do not have an IDL development license. The following chapters describe the process of packaging an application to run in the IDL Virtual Machine and packaging a Callable IDL application. Much of the information in these chapters is relevant whether or not your application end users have an IDL development license, but the assumption is that your end user will not have such a license.

# Types of IDL Runtime Applications

IDL applications can be written in IDL itself and distributed in IDL SAVE files, or they can be written in another programming language and distributed in a compiled binary format. IDL applications fall into the following two broad categories:

- **Native IDL applications** — A native IDL application is written entirely in IDL and saved in a SAVE file or series of SAVE files that can be restored and run by an IDL distribution.

  The process of creating applications written in IDL is the topic of this manual. This chapter describes the steps necessary to create and distribute an IDL application that uses a runtime or embedded license. Chapter 24, "Distributing Virtual Machine Applications" describes the steps necessary to package and distribute an IDL application that runs in the IDL Virtual Machine.

- **Callable IDL applications** — A Callable IDL application is written in another programming language, such as C or C++, and calls IDL as a subroutine. The process of creating Callable IDL applications is covered in the *External Development Guide*. Chapter 25, "Distributing Callable IDL Applications" describes the steps necessary to package and distribute a Callable IDL application.

# Licensing Options for IDL Runtime Applications

When you have an application that uses IDL and you want to distribute it to users who do not have an IDL development license, you have the following choices:

- Ask your users to install the free *IDL Virtual Machine* and run your application in the Virtual Machine

- Purchase a *runtime* or *embedded* license from IDL that enables you to bundle IDL with your application

- If your end user has an existing runtime license for another application, your application can run using that license

## Free Runtime License (IDL Virtual Machine)

The IDL Virtual Machine is a runtime version of IDL that can execute IDL SAVE files without an IDL license. Users install the IDL Virtual Machine with the IDL Installer available on an IDL distribution CD-ROM or from the IDL download Web site.

See Chapter 24, "Distributing Virtual Machine Applications" for additional details.

# Purchased Runtime and Embedded Licenses

You can purchase *runtime mode* licenses from IDL. Runtime mode licenses provide a way for you to include a licensed IDL installation with your IDL application or Callable IDL application. There are two types of runtime mode licenses available: *runtime* licenses and *embedded* licenses.

When you distribute a licensed version of IDL with your application, you provide your users with IDL functionality, but do not provide access to the IDL command line, the IDL Workbench, or the ability to compile IDL .pro files. Runtime and embedded licenses are appropriate for:

- Vertical-market packages developed in IDL but which appear to the user as stand-alone applications

- Software designed for use by operators or technicians who do not need programmatic access to IDL's full range of analytical tools

- Situations in which the you do not want end users to be able to modify functions written in the IDL language

- Organizations with existing investments in IDL code, where some mixture of distributable and development IDL licenses may be cost-effective

If your users need access to the full scope of IDL's features or advanced analytical tools outside the scope of your application, you might choose to distribute your application with an IDL development license. Contact your sales representative to purchase copies that you can distribute.

## Runtime Licensing

A runtime license enables a single user to run IDL SAVE files or Callable IDL applications. Runtime licenses require that you have some advance information about your users's computer. Runtime licenses come in two varieties:

- Node-locked licenses that can be installed only on a single machine

- Floating licenses that can be installed on any machine

See "Runtime Licensing" on page 573 for details.

## Embedded Licensing

An embedded license allows you to build license information into IDL SAVE files or Callable IDL applications. Embedded licenses do not require you to have advance information about your users' computers. See "Embedded Licensing" on page 577 for details.

# Limitations of Runtime Applications

IDL applications that run without an IDL development license — whether native IDL or Callable IDL — do not have access to the IDL compiler and thus cannot compile IDL source code from `.pro` files. As a result, operations that require the compiler will not execute when a development license is not present. In addition, if you are writing an IDL application to be distributed to users who do not have a development IDL license, you should be aware of the following limitations.

**Note** ───────────────────────────────────────────────
Since runtime applications do not provide access to the IDL command line, startup files are not executed. See "Understanding When Startup Files are Not Executed" for details.
────────────────────────────────────────────────────────

## Error Handling

Because the ON_ERROR procedure has the potential to force the IDL interpreter into an idle state when an error is encountered, use the CATCH procedure instead if your application will be distributed to users without a development IDL license.

## Working Directory of Runtime Applications

When a SAVE file is executed with a runtime or embedded license, IDL's current working directory will be the directory that contains the SAVE file.

## IDL Help

Support for the IDL 7.0 help system is not included in a runtime distribution by default. This means that applications that use the ONLINE_HELP procedure to display IDL help topics will fail unless you explicitly include the required support. If you use the MAKE_RT procedure to create a runtime distribution, you can use the IDL_HELP keyword to include the necessary files. See Chapter 22, "Providing Online Help For Your Application" (*Application Programming*) for additional discussion of the IDL 7.0 help system.

Support for the IDL Assistant help viewer (IDL's standard help viewer for releases 6.2 thorugh 6.4) is not included in a runtime distribution by default. If your application uses an IDL Assistant help system (that is, if it includes a `.adp` file), you will need to explicitly include the IDL Assistant help viewer. If you use the MAKE_RT procedure to create a runtime distribution, you can use the IDL_ASSISTANT keyword to include the necessary files.

# Steps to Distribute a Runtime Application

To create and distribute an IDL runtime application, do the following:

1. Create your application using an IDL development license. Test the application using the type of license you expect your end user to have.

2. If your application uses Callable IDL, see Chapter 25, "Distributing Callable IDL Applications" for information on creating a runtime distribution.

3. Decide on a licensing mechanism for your application. (For an overview of licensing mechanisms, see "Licensing Options for IDL Runtime Applications" on page 565.) If you choose to distribute an application that will run in the free IDL Virtual Machine, see Chapter 24, "Distributing Virtual Machine Applications" for information on creating a runtime distribution.

4. Obtain licenses for your application from IDL. See "Runtime Licensing" on page 573 or "Embedded Licensing" on page 577 for details.

5. Create an application distribution as described in "Creating an Application Distribution" on page 578.

6. Create invocation and use instructions for your application. See "Starting a Runtime Application" on page 579 for additional information.

7. Create an installer, if desired, and installation instructions for your application. See "Installing Your Application" on page 582 for additional information.

# Preferences for Runtime Applications

IDL's preference system allows developers, administrators, and individual users to control default values for many aspects of IDL's environment and configuration. Creators of runtime applications can take advantage of the preference system to customize the environment in which a particular application runs.

**Note** ———————————————————————————————————————

Before attempting to use preferences to customize the runtime IDL environment, you should have a clear understanding of how IDL loads and uses preference values. See Appendix E, "IDL System Preferences" (*IDL Reference Guide*) for a detailed discussion of the preference system.

———————————————————————————————————————————

## Preference Etiquette

IDL's preference system routines PREF_SET and PREF_COMMIT provide programmatic control over the values of preferences saved in an individual user's preferences file. **In general, as an application author, you should not use these routines in IDL code.** Since preference values set in the user preference file persist between sessions, changes made by your application using these routines will affect your end user's IDL environment even when he or she is running other applications.

Preference files loaded at application startup provide a much more user-friendly mechanism for specifying preference values that apply only to your application. To use this mechanism, create a preference file that contains the preference values you wish to have in effect when your application runs, and include the name of the preference file in the command that launches your application via the -pref command-line option. (See "Command Line Options for IDL Startup" (Chapter 1, *Using IDL*) for details.)

## Loading Preference Values at Application Startup

IDL provides the following mechanisms for loading preference values when an IDL application starts:

- Specifying a preference file via the -pref command line option.
- Providing an idl.pref file located in the same directory as the IDL DLL file (Windows only).
- Specifying individual preference values specified as command line options.

- Specifying individual preference values via the values of corresponding system environment variables.

- Modifying the idl.pref file in the resource/pref subdirectory of the application distribution. This method is only useful if you are distributing an IDL distribution to support your application — you should *not* modify an existing idl.pref file in your end user's installed IDL distribution.

**Note** ───────────────────────────────────────────────────

These mechanisms change the value of the specified preference or preferences for the current IDL session only. Values are *not* written to the user's preference file.

───────────────────────────────────────────────────────────

## Specifying Preferences at the Command Line

Of these options the first — specifying a preference file via the -pref command line option — is the most general and user-friendly. By specifying the values for preferences used by your application in a separate, application-specific preferences file, you can both control IDL's runtime environment and provide your end users with a mechanism to tune the IDL environment themselves. If one of your end users can achieve better performance using a different preference value, all that user needs to do is alter the value in the preference file loaded at startup.

## Providing an idl.pref File (Windows Only)

The second option — providing an idl.pref file in the same directory as the IDL DLL — is only available under Microsoft Windows.

There are three Windows-only runtime preferences:

- IDL_WINRT_FILE allows you to specify the name of a save file to be run when IDL starts up

- IDL_WINRT_FILE_TYPE allows you to specify the licensing mode of a runtime application

- IDL_WINRT_ICON allows you to specify the name of an icon file to use with a runtime application

These preferences are honored only when the idlrt.exe executable is in use. Their values are described in detail in "Windows Runtime Preferences" (Appendix E, *IDL Reference Guide*).

### Specifying Individual Preference Files

Specifying individual preference values at the command line provides little advantage over specifying the name of a preference file, but may be useful if the number of preferences to be specified is small.

### Using Environment Variables

The technique of using environment variables to specify preference values can be useful, but should be used with caution. Setting an environment variable provides a relatively easy way for your end users to override your preference settings without the need to modify the preference file you distribute. Depending on how the value of an environment variable is specified, however, the value may persist between invocations of your application. As a result, end users might experience unexpected behavior in other IDL applications (or in IDL itself) if an environment variable specified for your applications is in effect when the other applications are run.

### Modifying the Default Preferences File

You should only modify the resource/pref/idl.pref file if you are creating and distributing your own runtime application distribution.

See Appendix E, "IDL System Preferences" (*IDL Reference Guide*) for a detailed discussion of these options.

## Examples

Suppose you have created an IDL runtime application named myapp that performs numerous CPU-intensive calculations that could potentially use multiple CPUs on a multiprocessor system. If you want to ensure that your application uses at most two CPUs, you could include the following setting in a preference file named myapp.pref:

```
IDL_CPU_TPOOL_NTHREADS : 2
```

On UNIX platforms, you could then invoke your runtime application with a command line something like the following:

```
idl -rt=/myapp/myapp.sav -pref=/myapp/myapp.pref
```

On Windows platforms, you could create a preference file containing the following:

```
IDL_CPU_TPOOL_NTHREADS : 2
IDL_WINRT_FILETYPE: 0
IDL_WINRT_ICON: c:\myapp\myapp.ico
```

These preference values specify the maximum number of CPUs, the need for a runtime license for your application, and the application icon. If you name the preferences file `idl.pref` and place it in the `bin/bin.`*`platform`* subdirectory of your application distribution (where *`platform`* is your platform-specific `bin` directory), IDL will load the preferences when a user double-clicks on the application icon.

# Runtime Licensing

A *runtime* license allows you to run an IDL application that cannot display the IDL Workbench or IDL command line and which cannot compile `.pro` files. This type of licensing offers developers who have smaller customer bases the opportunity to buy single distribution licenses as they are needed, paying a small fee for each license. The license is either a node-locked license tied to the specific machine on which your application will run (which means you will need to obtain information about your customer's machine), or a more costly but less restricted floating license that will run on any machine.

When using runtime licensing, you can distribute licenses to your users in two ways:

- If you wish to distribute a licensed application to each customer, you can perform the necessary licensing steps for each license you purchase and distribute a ready-to-run application to each customer. This saves your customers from having to perform the licensing themselves, but forces you to create separate distributions for each customer.

- If you would rather create a single unlicensed distribution that you can distribute to all your customers, you can purchase a license for each customer and provide that license along with the information necessary for the customer to license your application.

## Ensuring That Your License is Used

To ensure that your application will run with your runtime license and not in the IDL Virtual Machine, add code similar to the following to your application before preparing your application distribution:

```
isVM = LMGR(/VM)
IF isVM THEN BEGIN
   void = DIALOG_MESSAGE(['Please contact the author', $
      'for licensing instructions'])
   RETURN
ENDIF
```

## Obtaining and Installing Runtime Licenses

Runtime applications are licensed using either node-locked licenses or floating single-user licenses. Node-locked licenses are tied to the specific computer on which the application will run, while floating licenses will run on any computer.

To license your runtime application, do the following:

1. Get information about the specific computer on which your application will run. The process for retrieving the required information depends on the end user's operating system, as described below.

2. Send this information to ITT Visual Information Solutions. We will generate a license file and send it to you.

3. Install the license file in a `license` subdirectory in your application's distribution, or provide instructions to your end user describing how to install the license file.

## Custom Features

You can request that your own custom feature license be added to your runtime license. Using a custom feature license allows you to specify that your application will only run if the custom feature license is present. Contact your ITT Visual Information Solutions sales representative for information on adding custom features to your runtime license.

## Obtaining a Windows License

In order to obtain the information needed to generate a node-locked license file, your end user must run the application `lmtools.exe` on the machine for which your application will be licensed. If your end user has already installed an unlicensed copy of your application, he or she will have access to the `lmtools.exe` application. Otherwise, you will need to provide the end user with a copy of the `lmtools.exe` file, which can be found in the `bin/bin.`*platform* directory of your IDL distribution.

Provide your end user with the following instructions:

1. In order for `lmtools.exe` to be able to retrieve the correct information, your system must have a properly-configured network interface card installed.

2. Run the `lmtools.exe` application. The **Lmtools** dialog appears.

3. Select the **System Settings** tab.

4. Click the **Save HOSTID Info to a File** button, then save the information to your desktop with the file name `hostid.txt`.

5. Send the `hostid.txt` file saved in the previous step to your application vendor.

When your end user has provided you with the information obtained by `lmtools.exe`, email this information to register@ITTvis.com or fax the

information to ITT Visual Information Solutions at (303) 786-9909. If you did not purchase IDL directly from ITT Visual Information Solutions, send the file to your local distributor.

ITT Visual Information Solutions will then send you a license file called `license.dat`.

## Obtaining a UNIX License

In order to obtain the information needed to generate a node-locked license file, your end user must run the application `lmhostid` on the machine for which your application will be licensed. If your end user has already installed a copy of IDL, he or she will have access to `lmhostid` application.

If your end user does not already have an IDL installation, you can provide a copy of the `lmhostid` file, located in the `bin/bin.`*`platform`* directory of your IDL distribution where *`platform`* is the platform-specific `bin` directory. Note that you must provide the executable for the platform on which your end user will run IDL.

Provide the end user with the following instructions:

1. Execute the command `lmhostid`. If the user has an IDL installation, the `lmhostid` file can be found in the `bin` subdirectory of that installation. Text similar to the following will be displayed:

   ```
   The FLEXlm host ID of this machine is "80598a67"
   ```

2. Provide the host ID returned by `lmhostid`, along with the hostname of the machine to your application vendor. (To obtain the hostname, enter the command `hostname`.)

When your end user has provided you with the information returned by `lmhostid` and the hostname of the machine, e-mail this information to register@ITTvis.com or fax the information to ITT Visual Information Solutions at (303) 786-9909. If you did not purchase IDL directly from ITT Visual Information Solutions, send the file to your local distributor.

We will then send you a license file called `license.dat`.

# Installing the License File

Once you have received a `license.dat` file from ITT Visual Information Solutions, you must ensure that it is installed in a `license` subdirectory in your application's distribution. You can either:

- Create a custom distribution for each individual end user by placing the `license.dat` file in the `license` subdirectory of your application's

distribution tree prior to packaging it for the end user. Your end user will not need to perform any licensing steps manually. This is a good solution if you have a small number of end users.

• Create a single unlicensed distribution that you provide to all your end users along with instructions to place the license.dat file you provide separately in the license subdirectory. This is a good solution if you have a relatively large number of end users, since it removes the need to create a custom distribution for each end user.

## Caution: IDL_LMGRD_LICENSE_FILE and LM_LICENSE_FILE Environment Variables

By default, when your application runs, IDL searches for a directory named license that contains a file named license.dat. It will use the first valid license it encounters; if no licences are found, the application will either run in unlicensed mode or exit.

If the end user has defined either the IDL_LMGRD_LICENSE_FILE or the LM_LICENSE_FILE environment variable, IDL will check *only* the license files specified by the environment variable. This means that if the end user has defined either the IDL_LMGRD_LICENSE_FILE or the LM_LICENSE_FILE environment variable for any reason, IDL might not find your application's license file even if it is placed correctly in a license subdirectory of your distribution.

See "License Sources" (Chapter 8, *Installation and Licensing Guide*) for a discussion of how the licensing environment variables interact.

# Embedded Licensing

An *embedded* license allows your application to run without an IDL license. It can be distributed to multiple users and will run on any system supported by IDL. Licensing an IDL application with an embedded license is the simplest form of licensing.

In order to create applications with embedded licenses, you must purchase a special *IDL Developer's Kit* license from ITT Visual Information Solutions. The Developer's Kit license gives your copy of IDL the ability to automatically embed a license in your application's SAVE file. See "Creating an Application Distribution" on page 578 for information on embedding the license information in your application's SAVE file.

**Note** —————————————————————————————————————————————————
Licenses for Callable IDL applications are embedded directly in the application code. See Chapter 25, "Distributing Callable IDL Applications" for details.
————————————————————————————————————————————————————————————

## Optional Embedded Features

When you purchase an IDL Developer's Kit license from ITT Visual Information Solutions, you can request that one or more *optional features* be included in the license. Optional feature licenses control access to additional-cost IDL modules, such as the IDL DICOM toolkit or the IDL DataMiner.

When your application attempts to use an additional-cost IDL module, IDL first checks to see if a license for the module is included in your application's embedded license. If no license for the module is included in the embedded license, IDL will check any license.dat files located in license directories in its search path, or in files specified by the LM_LICENSE_FILE environment variable. (See "Caution: IDL_LMGRD_LICENSE_FILE and LM_LICENSE_FILE Environment Variables" on page 576 for additional information about how IDL uses this environment variable.) If no license for the module is available, attempts to use that module's features will not succeed.

# Creating an Application Distribution

If your IDL application is intended to be run in an installation with an IDL development license, you do not need to create an IDL distribution. Simply package up your application files (either .pro files or .sav files, and any necessary data files) and distribute it to your users along with instructions describing how to install and launch the application.

If your application will be run by users who do not already have an IDL installation, or who do not have the proper IDL version, you can create and distribute a runtime application distribution. Runtime distributions are created using the MAKE_RT procedure; the process is described in detail in Chapter 26, "Creating a Runtime Distribution".

# Starting a Runtime Application

You must provide your end users with instructions describing how to start your application. You may choose to provide users with the name and location of your application executable along with a launch command to execute, or (if you are using an installer for your application) with shell scripts, shortcuts, or Start menu items.

The application startup process differs depending on whether you are supplying an IDL runtime distribution (created using the IDL Project interface or the `make_rt` script) or are relying on your user to install a full (if potentially unlicensed) IDL distribution. The following sections describe the process for each of these situations.

## Using an IDL Runtime Distribution

If you use the MAKE_RT procedure to create a runtime distribution, specifying a SAVE file for your application, application launch scripts for your application are created automatically. (You may need to modify the launch scripts.) If you do not use MAKE_RT, you can still create application launch scripts based on generic scripts included in the IDL distribution. See Chapter 26, "Creating a Runtime Distribution" for complete details.

## Using an Existing IDL Distribution

If you are relying on your end user to install an IDL distribution (licensed or not) before running your application, you can either give your users instructions based on the following information or create scripts to launch your application. The specifics depend on your end user's platform.

### Windows

To start a runtime application if you are not providing a runtime IDL distribution, either change directories to *IDL_DIR*\bin\bin.*platform* directory (where *IDL_DIR* is the main IDL directory and *platform* is the platform-specific bin directory) or ensure that this directory is included in the Windows PATH environment variable. Do one of the following:

- If your application runs in the IDL Virtual Machine, enter the following:

      idlrt -vm=<path><filename>

- If your application uses a runtime license, enter the following:

      idlrt <path><filename>

- If your application uses an embedded license, enter the following:

```
idlrt -em=<path><filename>
```

where *<path>* is the path to the SAVE file, and *<filename>* is the name of the SAVE file.

To simplify startup of your application, you can use the Windows launch script described in "Runtime Application Launch Scripts" in Chapter 26. Alternately, you can create a batch file that changes directories to the IDL bin directory and invokes idlrt with the SAVE file name. Such a batch file might look something like the following:

```
@ECHO OFF
REM This batch file launches the IDL runtime application myapp
cd C:\ITT\IDL70\bin\bin.x86
idlrt C:\mydir\myapp.sav
```

## UNIX / Macintosh

To start a runtime application if you are not providing a runtime IDL distribution, first ensure that the environment variable IDL_DIR is set to the path to the main directory of the IDL installation. For example, if IDL is installed in /usr/local/itt/idl70, you would set the IDL_DIR environment variable to this value. When the IDL_DIR environment variable is set, do one of the following:

*   If your application runs in the IDL Virtual Machine, enter the following:

    ```
    idl -vm=<path><filename>
    ```

*   If your application uses a runtime license, enter the following:

    ```
    idl -rt=<path><filename>
    ```

*   If your application uses an embedded license, enter the following:

    ```
    idl -em=<path><filename>
    ```

where *<path>* is the path to the SAVE file, and *<filename>* is the name of the SAVE file.

To simplify startup of your application, you can use the UNIX or Macintosh launch script described in "Runtime Application Launch Scripts" in Chapter 26. Alternately, you can create a shell script that sets the IDL_DIR environment variable and calls IDL with the correct flag and SAVE file name. Such a script might look something like the following:

```
#!/bin/sh
# This script launches the IDL runtime application myapp
IDL_DIR=/usr/local/itt/idl71
idl -rt=/mydir/myapp.sav
```

# Specifying Application Preferences at Startup

You can specify the values of IDL preferences in your startup command by including either the -pref command line option or by specifying individual preference values on the command line.

For example, suppose your application is installed in the directory /mydir. To have IDL load the preference values contained in a file named myapp.pref in the same directory when the application starts, you might modify your the UNIX startup script described above to read:

```
#!/bin/sh
# This script launches the IDL runtime application myapp
IDL_DIR=/usr/local/itt/idl71
idl -rt=/mydir/myapp.sav -pref=/mydir/myapp.pref
```

Similarly, to force a Windows runtime application to use software rendering, you could explicitly specify the preference value in a batch file that starts the application:

```
@ECHO OFF
REM This batch file launches the IDL runtime application myapp
cd C:\ITT\idl71\bin\bin.x86
idlrt C:\mydir\myapp.sav -IDL_GR_WIN_RENDERER 1
```

See "Preferences for Runtime Applications" on page 569 for details.

# What Happens When IDL Runs Your Application

When you launch an IDL runtime application, IDL does the following:

- Restores the specified SAVE file, if one is specified at the command line or when creating the distribution via the IDL Project interface or make_rt script

- Under Microsoft Windows, if no SAVE file is specified, restores the SAVE file specified by the IDL_WIN_RT preference

- If no SAVE file is specified, restores the file runtime.sav

IDL then calls the main procedure. This is one of the following:

- a procedure named main in the restored SAVE file

- a procedure with the same name as the .sav file

When the main procedure returns, IDL exits.

# Installing Your Application

Installation of your application on the end user's machine can be performed manually by the user, or it can be automated using an installer. There are a number of commercial applications available to help you build installers.

In order to avoid any possible conflicts with existing versions of IDL, you should warn your users NOT to install your application in the same directory as IDL x.x, where IDL x.x is the version used by your application.

**Note** ——————————————————————————————————————————————

ITT Visual Information Solutions' Global Services group can create installation packages for your application. Contact your ITT Visual Information Solutions sales representative for additional information.

————————————————————————————————————————————————————————

# Chapter 24
# Distributing Virtual Machine Applications

This chapter describes the process of creating IDL Virtual Machine applications for distribution.

# What Is a Virtual Machine Application?

The IDL *Virtual Machine* is a runtime version of IDL that can execute IDL `.sav` files without an IDL license. It is designed to provide IDL users with a simple, no-cost method for distributing IDL applications. It runs on all IDL-supported platforms, and does not require a license to run. This utility allows you to easily distribute IDL SAVE files to your colleagues or your customers, without requiring them to own an IDL runtime license.

Beginning with IDL 6.0, the IDL Virtual Machine is included with all IDL distributions. For the benefit of developers who need to debug applications designed to run in this environment, the IDL Virtual Machine can be started explicitly. Otherwise, if a SAVE file program is run without an IDL license, IDL defaults to the IDL Virtual Machine mode.

## If You Are *Running* a Virtual Machine Application

If you have received an IDL Virtual Machine application from someone else and are interested in running it on your own computer, do the following:

1.  **Install IDL.** If the application you received does not include a runtime IDL distribution or installer, you can use an IDL installer from ITT Visual Information Solutions. You do not need an IDL license to run Virtual Machine applications.

2.  **Install the Application.** Follow the application developer's instructions to install the Virtual Machine application on your computer.

3.  **Run the Application.** Follow the application developer's instructions to start the application, or see "Starting a Virtual Machine Application" on page 592.

## If You Are *Creating* a Virtual Machine Application

If you are creating an IDL Virtual Machine application, you should be familiar with the entire contents of this chapter. You may also wish to familiarize yourself with Chapter 23, "Distributing Runtime Mode Applications".

# Limitations of Virtual Machine Applications

The IDL Virtual Machine will run a compiled IDL SAVE file even if no IDL license is present. ITT Visual Information Solutions' aim with the IDL Virtual Machine is to facilitate IDL code collaboration and application distribution.

In addition to the limitations described in "Limitations of Runtime Applications" in Chapter 23, applications that run in the IDL Virtual Machine have the following restrictions:

- The IDL Virtual Machine displays a splash screen on startup.

- SAVE files must be created using IDL version 6.0 or later.

- No access to the IDL command line or IDL compiler is provided. Startup files are only executed when a command line is present. See "Understanding When Startup Files are Not Executed" (Chapter 1, *Using IDL*) for details.

- The use of the IDL EXECUTE function is disabled. (In most cases, calls to the EXECUTE function can be replaced with calls to the CALL_FUNCTION and CALL_PROCEDURE routines.)

- The Execute, GetVar, and SetVar methods to the IDL_IDLBridge object are disabled.

- The COM and Java IDL Export Bridge connector objects are disabled.

- Callable IDL applications will not run in the IDL Virtual Machine.

# Steps to Distribute Your Application

To create and distribute an IDL Virtual Machine application, do the following:

1. Create your application using an IDL development license, observing the limits described in "Limitations of Virtual Machine Applications" on page 585. Test the application in the IDL Virtual Machine.

2. Create one or more SAVE files containing your application. See "Creating Application SAVE Files" on page 589 for details.

3. Provide your users with instructions for installing an unlicensed copy of IDL, or create an application distribution as described in "Creating a Virtual Machine Distribution" on page 591.

4. Provide your users with instructions for installing your IDL application.

5. Provide your users with instructions for running your IDL application in the IDL Virtual Machine. See "Starting a Virtual Machine Application" on page 592.

# Preferences for Virtual Machine Applications

IDL's preference system allows developers, administrators, and individual users to control default values for many aspects of IDL's environment and configuration. Creators of runtime applications can take advantage of the preference system to customize the environment in which a particular application runs.

See "Preferences for Runtime Applications" in Chapter 23 for a discussion of using preferences in the context of any IDL runtime application, including applications that run in the IDL Virtual Machine.

The process of specifying preferences for a Virtual Machine application is complicated by the following facts:

• Since you are relying on a standard IDL Virtual Machine distribution rather than a distribution you create, it is more difficult to install a preferences file in the application distribution.

• On Microsoft Windows and Macintosh platforms, users may launch your Virtual Machine application by clicking on the SAVE file icon, or by dragging the SAVE file icon onto the Virtual Machine icon. This prevents you from specifying preferences via a command line option.

## Options for Windows Applications

If your Virtual Machine application runs under Microsoft Windows, you have the following options:

• Have your users launch the Virtual Machine application via the Windows command line, and use the `-pref` command line option to specify a preferences file or specify individual preferences.

• Have your users install an `idl.pref` file in the *IDL_DIR*/bin/bin.*platform* directory where *platform* is the platform-specific `bin` directory, and then launch the application by clicking on the SAVE file icon or by dragging it to the Virtual Machine icon.

• Instruct your users to set environment variables that correspond to the preferences you need to specify.

• If you are providing a runtime distribution for your application, you can install an `idl.pref` file in the *IDL_DIR*/bin/bin.*platform* directory yourself.

# Options for UNIX/Macintosh Applications

If your Virtual Machine application runs under UNIX (including Mac OS X), you have the following options:

- Have your users launch the Virtual Machine application via the shell command line, and use the `-pref` command line option to specify a preferences file or specify individual preferences.

- Instruct your users to set environment variables that correspond to the preferences you need to specify.

# Creating Application SAVE Files

An IDL application created in IDL 6.0 or later that does not use the EXECUTE function can be saved in one or more SAVE files that will run in the IDL Virtual Machine. If an IDL application is to be run in the IDL Virtual Machine, it is not necessary to include an IDL distribution with the SAVE file because IDL Virtual Machine is installed on the user's machine. The SAVE file need only include your own code, creating a smaller file that is easier to distribute.

To create SAVE files to run in the IDL Virtual Machine, do one of the following:

- Create SAVE files from one or more compiled `.pro` files with the SAVE procedure. See "Creating SAVE Files of Program Files" on page 56 for details, and refer to "SAVE" (*IDL Reference Guide*).

- Create SAVE files from a project by selecting **Project → Export** with the **Save File (`.sav`)** option specified.

**Note** —————————————————————————————

Creating SAVE files of object-oriented programs requires the use of RESOLVE_ALL with the CLASS keyword.

————————————————————————————————

## Single vs. Multiple SAVE Files

There are several ways to include the necessary routines in your application:

- For a native IDL application, include all routines in the main SAVE file that is restored when your application is started. This makes all routines available without having to restore any additional SAVE files, and reduces the number of SAVE files used by your application. The easiest way to do this is to add all `.pro` files to a project, and build the project.

- Create a separate SAVE file containing all your routines. You might use this method for a Callable application, if you want to keep certain routines separate from your main SAVE file in a native IDL application, or if your application includes routines provided to you as a SAVE file by another developer. To run any routines included in this SAVE file, you must restore the SAVE file by either calling a routine with the same name as the `.sav` file or restore it explicitly using RESTORE.

- Create a separate SAVE file for each routine used by your application. Assuming each `.sav` file uses the same name as the procedure or function it contains, this allows you to call each routine without having to explicitly

restore its SAVE file because IDL will search the current directory and the
defined !PATH for the `.sav` file and restore it automatically when it
encounters the first call to the routine.

# Version Compatibility of SAVE Files

The IDL Virtual Machine will execute IDL routines stored in SAVE files created with
IDL version 6.0 and later. Any SAVE files created with previous versions of IDL
must be recompiled using IDL 6.0 or later for them to run with the Virtual Machine.

# Creating a Virtual Machine Distribution

If your IDL Virtual Machine application is intended to be run by users who have IDL installed (with or without an IDL license), you do not need to create an IDL distribution. Simply package up your application files (.sav files and any necessary data files) and distribute them to your users along with instructions describing how to install and launch the application.

If your application will be run by users who do not already have an IDL installation, or who do not have the proper IDL version, you can create and distribute a runtime application distribution. Runtime distributions are created using the MAKE_RT procedure; the process is described in detail in Chapter 26, "Creating a Runtime Distribution".

# Starting a Virtual Machine Application

Installations of IDL that have access to a development license can create compiled binary versions of IDL applications; these compiled versions are stored in files with the extension `.sav`. Many applications stored in `.sav` files can be executed by the IDL Virtual Machine.

If you use the MAKE_RT procedure to create a runtime distribution, specifying a SAVE file for your application, application launch scripts for your application are created automatically. (You may need to modify the launch scripts.) If you do not use MAKE_RT, you can still create application launch scripts based on generic scripts included in the IDL distribution. See Chapter 26, "Creating a Runtime Distribution" for complete details.

Alternately, you can provide instructions for your users detailing how to run a `.sav` file in the IDL Virtual Machine. The process depends on your operating system:

- Windows
- UNIX
- Mac OS X

## Windows

Windows users can double-click on the `.sav` file, launch the IDL Virtual Machine and open the `.sav` file, or launch the `.sav` file in the IDL Virtual Machine from the command line.

### Double-Click a .sav File

To run an application stored in a .sav file, simply double-click on the .sav file icon in the Windows Explorer. If a development license is present, the application will run in a licensed copy of IDL. If no license is present, the IDL Virtual Machine window will open; click anywhere in the window to run the application in the IDL Virtual Machine.

### Launch the IDL Virtual Machine

To open a `.sav` file from within the IDL Virtual Machine:

1. Launch the IDL Virtual Machine and display the IDL Virtual Machine window by selecting **Start → Programs → IDL 7.1 → IDL Virtual Machine**.

2. Click anywhere in the IDL Virtual Machine window to close the window and display the file selection menu.

3. Locate and select the .sav file, and double-click or click **Open** to run it.

### Running from the Windows Command Line

To run a .sav file from the command line prompt:

1. Open a command line prompt. Select **Run** from the **Start** menu, and enter cmd.

2. Change directory (cd) to the *IDL_DIR*\bin\bin.*platform* directory where *platform* is the platform-specific bin directory.

3. Enter the following at the command line prompt:

```
idlrt -vm=<path><filename>
```

where *<path>* is the path to the .sav file, and *<filename>* is the name of the .sav file.

# UNIX

UNIX users must launch the IDL Virtual Machine from the UNIX command line.

To run a .sav file in the IDL Virtual Machine:

1. Enter the following at the UNIX command line:

```
idl -vm=<path><filename>
```

where *<path>* is the complete path to the .sav file and *<filename>* is the name of the .sav file. The IDL Virtual Machine window is displayed.

2. Click anywhere in the IDL Virtual Machine window to close the window and run the .sav file.

To launch the IDL Virtual Machine and use the file selection menu to locate the .sav file to run:

1. Enter the following at the UNIX command line:

```
idl -vm
```

The IDL Virtual Machine window is displayed.

2. Click anywhere in the IDL Virtual Machine window to close the window and display the file selection menu.

3. Locate and select the .sav file and click **OK**.

# Mac OS X

Macintosh users can launch the IDL Virtual Machine and open the `.sav` file, or launch the `.sav` file in the IDL Virtual Machine from the command line.

## Using the IDL Virtual Machine Icon

To open a `.sav` file from the IDL Virtual Machine:

1. Double-click the **IDL 7.1 Virtual Machine** icon to display the IDL Virtual Machine window.

2. Click anywhere in the IDL Virtual Machine window to close the window and display the file selection menu.

3. Locate and select the `.sav` file and click **OK**.

## Running from the Command Line

To run the IDL Virtual Machine from the UNIX command line:

1. Enter the following at the UNIX command line:

   ```
   idl -vm=<path><filename>
   ```

   where *<path>* is the complete path to the `.sav` file and *<filename>* is the name of the `.sav` file. The IDL Virtual Machine window is displayed.

2. Click anywhere in the IDL Virtual Machine window to close the window and run the `.sav` file.

To launch the IDL Virtual Machine and use the file selection menu to locate the `.sav` file to run:

1. Enter the following at the UNIX command line:

   ```
   idl -vm
   ```

   The IDL Virtual Machine window is displayed.

2. Click anywhere in the IDL Virtual Machine window to close the window and display the file selection menu.

3. Locate and select the `.sav` file and click **OK**.

# Chapter 25
# Distributing Callable IDL Applications

This chapter describes the process of creating Callable IDL applications for distribution.

# What Is a Callable IDL Application?

A *Callable IDL* application is written in another programming language, such as C or C++, and calls IDL as a subroutine. The process of creating Callable IDL applications is described in the *External Development Guide*.

Unlike applications written entirely in IDL, the process of creating an application distribution for a Callable IDL application is the same whether the application's end user has an IDL development license or not. This chapter describes the packaging process for Callable IDL applications using any licensing mechanism.

Callable IDL applications are packaged for distribution in much the same way as native IDL applications. Before beginning the process of packaging your Callable IDL application, you should be familiar with the contents of Chapter 23, "Distributing Runtime Mode Applications". This chapter describes the *additional* steps necessary to create and distribute a Callable IDL application.

## Licensing Options for Callable IDL Applications

When you have a Callable IDL application that you want to distribute to users who do not have an IDL development license, you must purchase a *runtime* or *embedded* license from ITT Visual Information Solutions. These options are described in detail in "Runtime Licensing" and "Embedded Licensing" in Chapter 23, "Distributing Runtime Mode Applications".

If your end user already has an IDL development license, you can simply package your Callable IDL application as described in this chapter and distribute it without including a license.

# Limitations of Runtime Mode Callable IDL Applications

IDL applications that run without an IDL development license — whether native IDL or Callable IDL — do not have access to the IDL compiler and thus cannot compile IDL source code from `.pro` files. As a result, operations that require the compiler will not execute when a development license is not present. In addition, if you are writing an IDL application to be distributed to users who do not have an IDL development license, you should be aware of the restrictions described in "Limitations of Runtime Applications" in Chapter 23, "Distributing Runtime Mode Applications".

**Note** ———————————————————————————————————

Startup files are not executed when you launch an IDL application without a command line. See "Understanding When Startup Files are Not Executed" (Chapter 1, *Using IDL*) for details.

———————————————————————————————————————————————

# Steps to Distribute a Callable IDL Application

To create and distribute a Callable IDL application, do the following:

1. Create your application using an IDL development license. Test the application using the type of license you expect your end user to have. See the *External Development Guide* for information on creating Callable IDL applications.

2. Decide on a licensing mechanism for your application. (For an overview of licensing mechanisms, see "Licensing Options for IDL Runtime Applications" in Chapter 23.)

3. Obtain licenses for your application from ITT Visual Information Solutions. See "Runtime Licensing" on page 600 or "Embedded Licensing" on page 601 for details.

4. Create an application distribution as described in "Creating a Callable IDL Application Distribution" on page 603.

5. Create invocation and use instructions for your application. See "Starting a Callable IDL Application" on page 606 for additional information.

6. Create an installer, if desired, and installation instructions for your application. See "Installing Your Callable IDL Application" on page 607 for additional information.

# Preferences for Callable IDL Applications

IDL's preference system allows developers, administrators, and individual users to control default values for many aspects of IDL's environment and configuration. Creators of runtime applications can take advantage of the preference system to customize the environment in which a particular application runs.

See "Preferences for Runtime Applications" in Chapter 23 for a discussion of using preferences in the context of a IDL runtime application.

The process of specifying preferences for a Callable IDL application is complicated by the fact that users never launch IDL directly. This means that in order to specify preference values, you must do one of the following:

- Modify the idl.pref file in the resource/pref subdirectory of the application distribution. This method is only useful if you are distributing an IDL distribution to support your application — you should *not* modify an existing idl.pref file in your end user's installed IDL distribution.

- Instruct your users to set environment variables that correspond to the preferences you need to specify, or explicitly set the variables yourself in a startup script or batch file.

# Runtime Licensing

A *runtime* license allows you to run an IDL application that cannot display the IDL Workbench or IDL command line and which cannot compile .pro files. This type of licensing offers developers who have smaller customer bases the opportunity to buy single distribution licenses as they are needed, paying a small fee for each license. The license is either a node-locked license tied to the specific machine on which your application will run (which means you will need to obtain information about your customer's machine), or a more costly but less restricted floating license that will run on any machine of a given platform.

**Note** ────────────────────────────────────────────────────

It is beyond the scope of this manual to discuss the creation of Callable IDL applications. See Chapter 16, "Callable IDL" (*External Development Guide*) for details. Note that applications using a runtime license must set the **IDL_INIT_RUNTIME** option when calling the **IDL_Initialize()** function, and must call **IDL_RuntimeExec()** rather than **IDL_Exec()**.

────────────────────────────────────────────────────────────────

When using runtime licensing, you can distribute licenses to your users in two ways:

- If you wish to distribute a licensed application to each customer, you can perform the necessary licensing steps for each license you purchase and distribute a ready-to-run application to each customer. This saves your customers from having to perform the licensing themselves, but forces you to create separate distributions for each customer.

- If you would rather create a single unlicensed distribution that you can distribute to all your customers, you can purchase a license for each customer and provide that license along with the information necessary for the customer to license your application.

See "Obtaining and Installing Runtime Licenses" on page 573 for information on obtaining and installing runtime licenses for your Callable IDL application.

# Embedded Licensing

An *embedded* license allows your application to run without an IDL license. It can be distributed to multiple users and will run on any system supported by IDL. Licensing an IDL application with an embedded license is the simplest form of licensing.

**Note**

It is beyond the scope of this manual to discuss the creation of Callable IDL applications. See Chapter 16, "Callable IDL" (*External Development Guide*) for details.

In order to create applications with embedded licenses, you must purchase a special *IDL Developer's Kit* license from ITT Visual Information Solutions. If you specify that you will be distributing a Callable IDL application when you purchase your Developer's Kit license, ITT Visual Information Solutions will provide you with a license string and some initialization code to be embedded into your application code before the application's initial call to IDL.

## Obtaining Your Licensing Information

Contact ITT Visual Information Solutions for your license information. You will need to provide the following information:

- The license installation number for your embedded license. Note that this number is different from the installation number for IDL itself.
- Your company name.
- Application title (e.g., My App).
- Name of the application executable (e.g., myapp).
- IDL interface being called (Callable IDL).
- Calling program language (e.g., VB, C++, C, Fortran).

You will receive a text file containing a function that IDL uses to retrieve the licensing information.

# Modifying Your Application Code

After you receive your license information, make the following changes to your application code, in the module from which you are initializing IDL. These instructions assume your code is written in C.

1. Define the licensing information for your application. Although your licensing information is individualized, it will resemble the following:

```
/* Callable Application license for: myapp, My App */
/* License built for IDL Version 7.1 */
static char *initStr[] = {
   "12345678abcdabcd",
   "12345678abcdabcd",
   "12345678abcdabcd",
   "12345678abcdabcd",
   "12345678abcdabcd",
   "" };
```

2. Allocate the following structure in the callable application.:

```
IDL_INIT_DATA init_data;
```

3. Initialize the structure in the callable application before IDL initialization:

```
init_data.applic = initStr;
init_data.options |= IDL_INIT_APPLIC;
```

4. Initialize IDL with the following statement (all platforms):

```
if (!IDL_Initialize(&init_data))
   return(FALSE);
```

# Creating a Callable IDL Application Distribution

This section discusses the process of creating an application distribution that includes the files necessary to run IDL, allowing you to distribute your application to users who do not already have IDL installed.

First, see "Creating an Application Distribution" in Chapter 23 for information on creating an IDL application distribution. The steps you will take after creating the IDL runtime distribution depend on the platform on which your Callable IDL application will run.

## Windows

Once you have created an IDL runtime distribution, you must do the following:

1. Add your Callable IDL application executables to the bin/bin.*platform* subdirectory of the distribution where *platform* is the name of the platform for which you created the application.

2. If your application uses preferences, edit the resource/pref/idl.pref file to contain the correct preference values.

3. If you are using the launch script generated by the MAKE_RT procedure, modify the launcher's .ini file to invoke your Callable IDL application rather than the idlrt.exe executable. Alternately, you can simply provide instructions detailing how to execute your Callable IDL executable file.

## UNIX

Once you have created an IDL runtime distribution, you must do the following:

1. Add your Callable IDL application executables to the bin.*platform* directory, where *platform* is the name of the platform for which you created the application. If you are distributing your application on multiple platforms, copy the executable for each platform to the corresponding bin.*platform* directory. Placing your executables in the bin.*platform* directory offers a couple of advantages:

   • It simplifies application startup, especially if your application is distributed for multiple platforms. The application startup script calls a script in the bin directory. This script is designed to start the correct executable, depending on the platform on which it is being executed. This

allows the user to start the application on any platform by simply executing the startup script in the top-level directory, thereby saving the user from having to know the directory in which the executable is located.

- It saves the user, or your installation script, from having to set the LD_LIBRARY_PATH environment variable because sharable libraries are located in the bin.*platform* directory.

2. Rename the idl script. The idl script is located in the bin directory of your distribution. For Callable IDL applications, this script *must* use the same name as your application executable in the bin.*platform* directory. For example, if your application executable in the bin.*platform* directory is called myapp, rename the idl script in the bin directory to myapp.

3. Edit the startup script. In the top-level directory of your application distribution, there is a startup script with the name specified by the startcommand parameter you specified when you ran the make_rt script. Make the following changes to this script:

   A. Edit the startup script to execute the script in the bin directory that you renamed in the previous step. For example, if your application executable in the bin.*platform* directory is called myapp, and you therefore renamed the idl script in the bin directory to myapp, you would edit the startup script in the top-level directory as follows:

   ```
   ./bin/myapp
   ```

   **Note**
   The above command requires the user to execute the startup script from the top-level directory of your application distribution. To allow the user to launch your application from a different directory, the user (or your installation script) could change the command to use the full path to the script in the bin directory. See the example after the following step.

   B. In order to allow your application to find the correct executable (either IDL or a Callable IDL executable), the IDL_DIR environment variable must be set on the user's machine to point to the top-level directory of your application. Because this location is not known until the user installs your application, IDL_DIR must be set by either an installation script or by the user.

   If there are other ITT Visual Information Solutions products installed on the user's machine, IDL_DIR may already be set. For this reason, IDL_DIR should be set for the instance of the shell that will be used to start your application, but should not be set in the user's login scripts such

as `.cshrc` or `.profile`. This allows IDL_DIR to be set properly for your application, without conflicting with the IDL_DIR setting for other products the user may have installed.

The most convenient way to set IDL_DIR on the user's machine is to have your installation script (or the user) edit the startup script. This saves the user from having to manually set IDL_DIR prior to launching your application. You can either provide the user with instructions on adding the necessary commands to the startup script, or you can have your installation script modify the startup script. For example, if an application called `myapp` is installed in the `/home/apps` directory, your startup script would resemble the following:

```
IDL_DIR=/home/apps
export IDL_DIR
/home/apps/bin/myapp
```

If you do not modify the startup script, the user must set IDL_DIR at the command prompt prior to launching your application. For example, if your application is installed in the user's `/home/myapp` directory, the user could execute the following command at the C shell prompt:

```
setenv IDL_DIR /home/myapp
```

4. If your application uses preferences, edit the `resource/pref/idl.pref` file to contain the correct preference values.

# Starting a Callable IDL Application

You must provide your end users with instructions describing how to start your application. You may choose to provide users with the name and location of your application executable along with a launch command to execute, or (if you are using an installer for your application) with shell scripts, shortcuts, or Start menu items. The specific instructions you provide will depend on your end user's platform.

## Windows

To start a Callable IDL application if you have exported an IDL runtime distribution using the MAKE_RT procedure, change directories to the `application\idldir\bin\bin.platform` directory (where `application` is the name of the directory that contains your exported distribution, `idldir` is the IDL directory inside your application directory, and `platform` is either `x86` or `x86_64`, depending on the verison of IDL your application relies on) and double-click on the executable file you created.

Alternately, you can use the Windows launch application described in

**Note** ─────────────────────────────────────────────────────────────
The executable file must reside in the `bin\bin.platform` subdirectory of your exported application distribution. For your users' convenience, you may want to create a Windows shortcut to the executable file in another location.
─────────────────────────────────────────────────────────────────────

## UNIX

To start a Callable IDL application if you have exported an IDL runtime distribution using the MAKE_RT procedure, change directories to the `application/idldir/bin` directory (where `application` is the name of the directory that contains your exported distribution and `idldir` is the IDL directory inside your application directory) and execute the renamed `idl` script file.

# Installing Your Callable IDL Application

Installation of your application on the end user's machine can be performed manually by the user, or it can be automated using an installer. There are a number of commercial applications available to help you build installers.

In order to avoid any possible conflicts with existing versions of IDL, you should warn your users NOT to install your application in the same directory as IDL x.x, where IDL x.x is the version used by your application.

**Note** ————————————————————————————————————————

ITT Visual Information Solutions' Global Services group can create installation packages for your application. Contact your ITT Visual Information Solutions sales representative for additional information.

————————————————————————————————————————

# Chapter 26

# Creating a Runtime Distribution

This chapter discusses the process of creating an application distribution that includes the files necessary to run IDL, allowing you to distribute your application to users who do not already have IDL installed.

# About Runtime Distributions

If your IDL application is intended to be run in an installation with an IDL development license, you do not need to create an IDL distribution. Simply package up your application files (either .pro files or .sav files, and any necessary data files) and distribute it to your users along with instructions describing how to install and launch the application.

If, however, you intend to distribute your application to users who do not have an existing IDL installation, or if you want your application to run directly from removable media such as a CD-ROM or DVD-ROM, you must create a *runtime distribution*. A runtime distribution contains the IDL executable files, dynamically-loaded library files, and resource files needed to execute an IDL application that has been packaged in an application .sav file.

In versions of IDL prior to version 7.0, there were two methods available to create a runtime distribution:

- using the **Export** feature of the IDL Development Environment's **Project** interface (Windows and UNIX)
- using the make_rt script (UNIX only)

In IDL 7.0, these methods were replaced by the MAKE_RT procedure. MAKE_RT provides a cross-platform mechanism for building runtime distributions for multiple platforms. The MAKE_RT procedure itself is described in the *IDL Reference Guide*; this chapter elaborates on some of the issues surrounding creating and distributing runtime applications.

# Creating a Distribution Using MAKE_RT

The MAKE_RT procedure creates an IDL distribution for one or more target platforms.

**Note** ————————————————————————————————————

You do not need to create a SAVE file in order to use MAKE_RT, but application launch scripts will only be created if a SAVE file is included.

————————————————————————————————————————————

To create a runtime distribution for your application, you will do the following:

1. Collect Required Information
2. Modify or Create a Manifest File
3. Run the MAKE_RT Procedure
4. Add Required Files to Your Distribution
5. Modify the Launch Scripts

## Collect Required Information

Before using the MAKE_RT procedure to create a runtime distribution, you will need to collect the following information and make several decisions about how your application will run. You will need to:

• Choose a name for your runtime application. The application name will be used as the name of the directory that contains the runtime distribution, and will be used as the base name for any launch scripts created by MAKE_RT.

• Know the full path to the output directory where your distribution will be created. This directory must exist and you must have the appropriate permissions to write files into it. A directory with the same name as your application will be created in the output directory.

• Know the full path to the SAVE file that contains your application code, if one exists. If you specify a SAVE file, launch scripts will be created to run the application it contains.

• Decide which platforms you want your application to run on. You must have access to an installed IDL distribution for every platform you want to include in your runtime distribution. (Note that you do not need to have licenses for all of the platforms; an installed distribution is all that is required.)

- Decide whether your application should use an IDL license, if one is available. By default, MAKE_RT will create launch scripts that will use an IDL license if one is present; if no license is present, the application will run in the IDL Virtual Machine. If you want to ensure that your application runs in the Virtual Machine even if an IDL license is available, specify the VM keyword to the MAKE_RT procedure.

- Know the full path to your custom manifest file, if you are using one. Manifest files are described in the following section.

# Modify or Create a Manifest File

The default manifest file, *IDL_DIR*/bin/make_rt/manifest_rt.txt (where *IDL_DIR* is the IDL installation directory) contains entries for all of the files necessary to create a runtime IDL distribution for all supported platforms. In most cases, you can use the manifest_rt.txt file without modification, and the MAKE_RT procedure will select the appropriate files to build the distribution you specify.

In some cases, however, you may need to modify or add to the list of files contained in manifest_rt.txt. For more on the format of this file, see "Working with the manifest_rt.txt File" on page 616.

# Run the MAKE_RT Procedure

Run the MAKE_RT procedure to create the runtime distribution. The syntax and options are described in the *IDL Reference Guide*.

## Creating Mixed UNIX/Windows Distributions

The MAKE_RT procedure allows you to create a single runtime distribution that supports multiple platforms. In order to created a mixed-platform distribution, MAKE_RT must have access to an IDL installation directory that contains all of the required files.

On UNIX platforms (Solaris, Macintosh OS X, and Linux), a single installation directory can contain files for multiple operating systems. If you are running IDL on a UNIX platform and wish to create a runtime distribution for one or more UNIX platforms (but not Microsoft Windows), MAKE_RT can create the distribution in a single operation. On Microsoft Windows platforms, an IDL installation directory can only contain Windows files.

If you want to create a runtime distribution that includes both Microsoft Windows and one or more UNIX platforms, you will need to run the MAKE_RT procedure at

least twice: once to create the Windows distribution and one or more additional times to create distributions for the UNIX platforms. You can use the same target directory for all invocations of MAKE_RT; any files that are duplicated in the selected platforms' distributions will be quietly overwritten.

For example, suppose you want to create a runtime distribution that supports 32-bit Windows, Macintosh OS X, and 32-bit Linux, and both 32- and 64-bit Solaris. IDL is installed on the Windows machine, on the Macintosh machine, and in a shared location for the Linux and Solaris machines. The process of creating a mixed runtime distribution would look something like this:

1. On the Windows machine, run IDL and give the following command:

   ```
   MAKE_RT, 'myApp', Outdir, SAVEFILE=sfile
   ```

   where *Outdir* is a directory on a network drive that is accessible to all systems, and *sfile* is the full path to the IDL SAVE file that comprises your application.

2. On the Macintosh, run IDL and give the following command:

   ```
   MAKE_RT, 'myApp', Outdir, SAVEFILE=sfile
   ```

   where *Outdir* is a directory on a network drive that is accessible to all systems, and *sfile* is the full path to the IDL SAVE file that comprises your application.

   **Note**
   Although your SAVE file has already been copied to the application directory, you must include the SAVEFILE keyword to MAKE_RT again here in order to create the Macintosh launch scripts for the application.

3. On either a Linux or a Solaris machine, run IDL and give the following command:

   ```
   MAKE_RT, 'myApp', Outdir, SAVEFILE=sfile, /LIN32, /SUN32,
   /SUN64
   ```

   where *Outdir* is a directory on a network drive that is accessible to all systems, and *sfile* is the full path to the IDL SAVE file that comprises your application.

   **Note**
   Again, you must include the SAVEFILE keyword to MAKE_RT here in order to create the UNIX launch script for the application.

# Add Required Files to Your Distribution

After you have created a distribution using MAKE_RT, any files that are not part of the IDL distribution, as well as any required IDL files that you did not add to the manifest, must be manually copied to your distribution. Do the following:

1. If your application requires any data files that are not in the IDL distribution, including ASCII, binary, or image files, add them to your distribution.

2. If your application includes more than one SAVE file, add the files to the distribution.

3. If your application includes help files or other documentation, add the files to the distribution.

For information on creating and restoring SAVE files, see Chapter 4, "Creating SAVE Files of Programs and Data".

# Modify the Launch Scripts

The MAKE_RT procedure will generate launch scripts for each of the platforms supported by your runtime distribution. The launch scripts are named with the string specified as the *Appname* argument to MAKE_RT, and several values within the launch scripts are modified.

If you specify a value for the SAVEFILE keyword, the launch scripts will automatically invoke the application contained in the SAVE file. If the SAVEFILE keyword is not present, the launch scripts will simply launch IDL in runtime or virtual machine mode, which presents a file selection dialog allowing the end-user to select a SAVE file to restore and execute.

**Note** ────────────────────────────────────────────────────────────────────
The launch scripts will function unmodified, but you may want to customize them.
────────────────────────────────────────────────────────────────────────────

The launch scripts are described in detail in "Runtime Application Launch Scripts" on page 618. This section describes some additional modifications you may want to make.

## Windows

The launch script for Windows platforms is named *Appname.exe*, where *Appname* is the value of the *Appname* argument to MAKE_RT. The launch script is configured via an initialization file named *Appname.ini*. You may want edit *Appname.ini* to modify the text used in the application launch dialog.

If your application runs on both 32- and 64-bit IDL, you may want to create separate buttons to launch the different versions.

See "Windows Launch Script" on page 618 for additional details.

## Macintosh

The launch script for Macintosh platforms is an AppleScript named *Appname.app*, where *Appname* is the value of the *Appname* argument to MAKE_RT.

See "Macintosh Launch Script" on page 621 for additional details.

## Linux/Solaris

The launch script for Linux and Solaris platforms is a shell script named *Appname*, where *Appname* is the value of the *Appname* argument to MAKE_RT.

See "Linux/Solaris Launch Script" on page 623 for additional details.

# Working with the manifest_rt.txt File

In many cases you can use the *IDL_DIR*/bin/make_rt/manifest_rt.txt file (where *IDL_DIR* is the IDL installation directory) without modification. If, however, your application uses files that are not part of the IDL distribution, or if you want to include features described in manifest_rt.txt but commented out of the default runtime distribution, you can create a custom manifest file. To create a custom manifest file, begin by copying the manifest_rt.txt file and giving your file a new name, such as manifest_custom.txt.

**Warning**
Use a text editor such as vi, emacs, TextEdit, or the Windows Notepad to edit manifest files. Blank lines and any text following a comment character (semicolon) will be ignored by the MAKE_RT procedure.

## Format of the Manifest File

The manifest files used by the MAKE_RT procedure are plain text files that contain one line for each file in the IDL installation directory that can potentially be copied to the runtime distribution.

Each entry in the manifest file corresponds to a file that exists in the IDL distribution for a particular platform. (Note that although files for all supported platforms are included in the manifest file, the MAKE_RT procedure only attempts to copy files for platforms specified when the procedure is run.)

In addition to editing the contents of the manifest file based on the keywords specified at runtime, the MAKE_RT procedure applies the following rules when creating its list of files to copy to the runtime distribution:

- blank lines are ignored
- lines that begin with a semicolon are ignored
- text following a semicolon that is not at the beginning of a line is ignored

## Removing IDL Features

Some sections of the manifest_rt.txt file are noted as optional. If your application does not use the features contained in one or more of these sections, you may be able to remove them from your custom manifest file, creating a smaller runtime distribution.

**Warning** ———————————————————————————————————————
If you choose to remove one or more optional features, be sure to test your application thoroughly using the runtime distribution. Removing an optional feature may reveal dependencies in your code of which you were unaware.

# Including Optional IDL Features

The manifest_rt.txt file includes sections that are commented out (that is, each line begins with a semicolon). These sections correspond to features (support for the IDL Dataminer, for example) that are rarely used or that require a special license. To include these features in your runtime distribution, you will need to edit the manifest file to remove the comment characters.

**Note** ———————————————————————————————————————
Even if you uncomment all of the entries for a particular feature, only the files that are required for the platforms you specify will be copied by MAKE_RT.

# Including Non-IDL Files

To include features that are not described in the manifest_rt.txt file, add new lines describing the location of the files. Note that paths specified in the manifest file are relative to *IDL_DIR*, and that files will have the same relative location with the IDL runtime distribution as they have in the source distribution.

**Note** ———————————————————————————————————————
When adding non-IDL files to a manifest file, use the forward slash ("/") as the directory separator, even on Windows platforms.

**Tip** ———————————————————————————————————————
If you are unable to place your extra files into the source IDL distribution, you may want to manually copy the files after the runtime distribution has been built, as described in "Add Required Files to Your Distribution" on page 614.

# Runtime Application Launch Scripts

The `bin/make_rt` subdirectory of the IDL installation directory contains generic launch scripts for Windows, Macintosh, and Linux/Solaris applications. If you use the MAKE_RT procedure to create a runtime distribution, the appropriate launch scripts will be copied to your distribution and renamed to match your application. If you specify a value for the SAVEFILE keyword, the launch scripts are modified by MAKE_RT to launch the specified SAVEFILE application. If the SAVEFILE keyword is not specified, executing the launch script prompts the end-user to select a `.sav` file to restore and run.

This section describes the different launch scripts in more detail, and explains how to configure and use them. Note that while some of the steps described here are performed by the MAKE_RT procedure, you may still need to modify the scripts to achieve the desired behavior.

## Windows Launch Script

To use the application launcher, follow the steps outlined below.

**Note** _____

If you use the MAKE_RT procedure, the launch scripts are copied to your runtime distribution and renamed to match the *Appname* argument automatically. The `start_app_win.ini` file is modified to run your SAVE file or open a file-selection dialog, depending on the value of the SAVEFILE keyword.

_____

### Copy and Rename the start_app_win.exe File

Copy the file

```
IDL_DIR\bin\make_rt\start_app_win.exe
```

(where *IDL_DIR* is your IDL installation directory) to the location of your runtime distribution. If you want, rename `start_app_win.exe` to reflect the name of your application. (Be sure to retain the `.exe` extension.) For example, if your application is named "HydroPlot," you could rename the `start_app_win.exe` file as `hydroplot.exe`.

### Copy and Rename the start_app_win.ini File

When a user clicks on the executable file (`start_app_win.exe` or whatever you have renamed it), the executable searches for and reads a `.ini` file with the same base name as the executable. If you renamed `start_app_win.exe`, you will also

need to rename the `.ini` file with the same base name. For example, if you renamed `start_app_win.exe` as `hydroplot.exe`, you would rename `start_app_win.ini` as `hydroplot.ini`.

Copy the file

```
IDL_DIR\bin\make_rt\start_app_win.ini
```

(where *IDL_DIR* is your IDL installation directory) to the location of your runtime distribution. Rename the `.ini` file to match the name of the executable file, if you have changed it from `start_app_win.exe`.

## Modify the start_app_win.ini File

The `.ini` file (`start_app_win.ini` or whatever you have renamed it) specifies what will happen when the user runs the `.exe` file. If you use the MAKE_RT procedure and specify a value for the SAVEFILE keyword, the `.ini` file is rewritten to launch your application. If you copy the `.ini` file manually, you must modify it as described below.

The `start_app_win.exe` file can either run a single application immediately or display a dialog with up to four buttons, each of which invokes a different application. The configuration of the dialog (including whether or not it is displayed at all) is controlled by the `.ini` file.

The `.ini` file contains five sections, one labelled `[DIALOG]` and four labelled `[BUTTON`*n*`]` (where *n* is a number between 1 and 4). The contents of each type of section are described below.

### DIALOG Section

```
[DIALOG]
Show=False
BackColor=&H6B1F29
Caption=<any string>
Picture=.\splash.bmp
DefaultAction=<path to application>
```

- **Show** — this field can contain the string `True` or the string `False`. If `Show=True`, the dialog is displayed, and the `DefaultAction` is *not* executed. If `Show=False`, the dialog is *not* displayed, and the `DefaultAction` is executed immediately when the user double-clicks on the `start_app_win.exe` icon.

- **BackColor** — this field contains an RGB color triplet specified in hexadecimal notation. This color will be used in any part of the dialog that is

*not* covered by the image specified as the value of the `Picture` field. To make the background white, set `BackColor=&HFFFFFF`.

*   **Caption** — this field contains a string that will be displayed in the title bar of the dialog, if `Show=True`.

*   **Picture** — this field contains the relative path to a Windows bitmap file that will be displayed in the dialog if `Show=True`. The image will be positioned with its upper left corner in the upper left corner of the dialog window. To completely fill the dialog, the image contained in the bitmap file should be 480 x 335 pixels. Any area of the dialog that is not filled by the image will be displayed in the color specified in the `BackColor` field.

*   **DefaultAction** — this field contains the command that should be executed when `start_app_win.exe` is run if `Show=False`. In most cases, you will need to specify the relative path to the `idlrt.exe` file in the IDL distribution on your CD-ROM, followed by the `-vm` flag and the relative path to your application's SAVE file.

    For example, if you have placed the SAVE file for the application `hydroplot.sav` in the `hydroplot` directory of the CD-ROM along with the `start_app_win.exe` application, the following `DefaultAction` launches `hydroplot.sav` in the IDL Virtual Machine when the user double clicks on the `start_app_win.exe` icon:

    ```
    DefaultAction=.\idl71\bin\bin.x86\idlrt.exe
    -vm=hydro\hydroplot.sav
    ```

    (The DefaultAction specification should be on a single line.)

### BUTTON Sections

There can be up to four `[BUTTON]` sections. The format is the same for any section of this type.

**Note** ────────────────────────────────────────────────────────────────

If the `Show` field of the `[DIALOG]` section is set to False, no buttons will be displayed, regardless of the content of the `[BUTTON]` sections.

────────────────────────────────────────────────────────────────────────

```
[BUTTON1]
Show=True
Caption=<any string>
Action=<path to application>
```

*   **Show** — this field can contain the string `True` or the string `False`. If `Show=True`, the button will be displayed on the dialog.

- **Caption** — this field contains a string that will be displayed on the button, if `Show=True`.

- **Action** — this field contains the command that should be executed when the user clicks on the button, if `Show=True`. See `Default Action` above for an explanation of the format of the command string.

  To create a button that simply closes the dialog without executing anything, set `Action=Exit` on the button.

### Copy and Modify the autorun.inf File

If you want your application to launch automatically when the user inserts your CD-ROM, you must modify the `autorun.inf` file. The `autorun.inf` file contains the following lines:

```
[autorun]
open = start_app_win.exe
icon = idl.ico
```

If you want your application to launch automatically when the user inserts the CD-ROM, copy the file

```
IDL_DIR\bin\make_rt\autorun.inf
```

(where *IDL_DIR* is your IDL installation directory) into your runtime distribution and modify the

```
open = start_app_win.exe
```

line to reflect the name of the executable file you want to launch automatically. For example, if you renamed `start_app_win.exe` to `hydroplot.exe`, change the line to read:

```
open = hydroplot.exe
```

If your executable file displays a dialog, you might want to modify the

```
icon = idl.ico
```

line to specify an icon that will be displayed in the Windows task bar. If you specify an icon file in your `autorun.inf` file, you must ensure that the icon file is included in the root directory of your CD-ROM.

# Macintosh Launch Script

The `bin/make_rt` subdirectory of the IDL installation directory includes two Applescripts that you can use to launch your application. To use the Applescripts, follow the steps outlined below.

**Note**

If you use the MAKE_RT procedure, the launch scripts are copied to your runtime distribution automatically. The start_app_mac.app file is renamed to match the *Appname* argument and modified to run your SAVE file or open a file-selection dialog, depending on the value of the SAVEFILE keyword. You can edit the .app file using the AppleScript editor.

A text version of the script named *Appname*_mac_script_source.txt is also saved in the same directory as the .app file. You can delete the .txt file or copy it into the AppleScript editor if you want to modify the script.

## Copy and Rename the Applescript Files

Use the Finder to copy the files

```
IDL_DIR/bin/make_rt/start_app_mac.app
IDL_DIR/bin/make_rt/Utils_applescripts.scpt
```

(where *IDL_DIR* is your IDL installation directory) to the location of your runtime distribution. If you want, rename start_app_mac.app to reflect the name of your application. For example, if your application is named "HydroPlot," you could rename the start_app_mac.app file as hydroplot.app. Do not rename Utils_applescripts.scpt.

## Modify the start_app_mac.app File

Use the Applescript editor to modify the value of the idlApp and idlDir variables in the start_app_mac.app file (or whatever you have renamed it) as shown below:

```
(*
Specify the path to the IDL SAVE file that launches the virutal
machine application, relative to the location of the script
*)
set idlApp to "my_app.sav" as string

(*
Specify the path to the top directory of the IDL distribution,
relative to the location of the script.
*)
set idlDir to "idl71" as string
```

where the IDL installation is in the directory idl71 and the application is in a SAVE file named my_app.sav.

**Note**

By default, the start_app_mac script will launch IDL in 64-bit mode if possible. To force IDL to launch in 32-bit mode (if, for example, your application

uses DICOM features or the IDL Advanced Math and Stats IMSL libraries), edit the line in the `start_app_mac.app` file that looks like

```
set idlCmd to IDLDirFolder & "bin/idl -vm=" & myAppPath
```

to include the `-32` flag as shown in the file comments.

# Linux/Solaris Launch Script

The `bin/make_rt` subdirectory of the IDL installation directory includes a bourne shell script that you can use to launch your application. To use the script, follow the steps outlined below.

**Note**

If you use the MAKE_RT procedure, the launch scripts are copied to your runtime distribution automatically. The `start_app_unix` file is renamed to match the *Appname* argument and modified to run your SAVE file or open a file-selection dialog, depending on the value of the SAVEFILE keyword.

## Copy and Rename the start_app_unix File

Copy the file

```
IDL_DIR/bin/make_rt/start_app_unix
```

(where *IDL_DIR* is your IDL installation directory) to the location of your runtime distribution. If you want, rename `start_app_unix` to reflect the name of your application. For example, if your application is named "HydroPlot," you could rename the `start_app_unix` file as `hydroplot`.

## Modify the start_app_unix File

Using a text editor, modify the value of the `idlapp` and `IDL_DIR` variables in the `start_app_unix` file (or whatever you have renamed it) as show below:

```
# Specify the path to the IDL SAVE file that launches
# the Virtual Machine application, relative to $topdir.
idlapp=$topdir/my_app.sav

# Specify the path to the top directory of the IDL
# distribution, relative to $topdir.
IDL_DIR=$topdir/idl71 ; export IDL_DIR
```

where the IDL installation is in the directory `idl71` and the application is in a SAVE
file named `my_app.sav`.

**Note** ─────────────────────────────────────────────────────────────────────

If you use the MAKE_RT procedure and specify a value for the SAVEFILE
keyword, the `start_app_unix` file is rewritten to launch your application.

─────────────────────────────────────────────────────────────────────────────

# Incorporating the IDL DataMiner

If your application uses IDL DataMiner, use the DATAMINER keyword to the MAKE_RT routine. You will also need to add some files and move other files before you distribute your application. The changes you make will depend on the operating system you are using.

## Windows

If your application uses IDL DataMiner, please call ITT Visual Information Solutions Technical Support for instructions.

- E-mail: support@ittvis.com
- Phone: (303) 413-3920

## UNIX

You must modify the odbc.ini file to include information about the drivers you are using. This file is located in the resource/dm/<*OS_NAME*> directory of the distribution tree you have just created. After modifying this file, it must be placed in each user's home directory. For details on the modifications you must make to the odbc.ini file, see the *IDL DataMiner* manual.

# Distributing Runtime Applications on Removable Media

The runtime distribution you create using MAKE_RT can be distributed on removable media such as a CD-ROM, DVD-ROM, or flash drive. The application will, in most cases, run directly from the removable media without any need to install files on the end-user's system. (See "Installation Issues: Windows" on page 626 for additional details.)

To distribute the runtime distribution on removable media, simply copy the entire directory hierarchy created by the MAKE_RT routine to the removable media. Users can then execute your launch scripts directly from the media.

For better performance, you may want to instruct your end-users to copy the runtime distribution to a local hard drive. Again, simply copying the entire directory hierarchy from the removable media to the hard drive is sufficient. Similarly, if you distribute the runtime application without physical media (using a downloadable archive file, for example), simply unpacking the archive file on a local hard drive should suffice to install your application.

**Note** ─────────────────────────────────────────────

Copying a runtime distribution onto the user's hard disk does not "install" IDL in the usual sense. No file associations or symbolic links are created.

─────────────────────────────────────────────────────

## Installation Issues: Windows

When you install IDL for Windows, the installation program ensures that all Microsoft Windows system libraries required by IDL are installed. If you are distributing a runtime application that will run on a Windows system that does not have an installed version of IDL, it is possible (although somewhat unlikely) that the required system libraries will *not* be present on your end-user's computer.

If your application does not run correctly on your user's machine, the missing system libraries may be the problem. The *IDL_DIR*/bin/make_rt directory includes two small installation programs that ensure the required system libraries are present. You are free to distribute these to your own users. Instruct your users to run the appropriate system library installer if they have problems:

- For 32-bit Windows systems, run systemdll32_setup.exe.
- For 64-bit Windows systems, run systemdll64_setup.exe.

**Note**

The system library installers will only install the required libraries if the correct versions (or later versions) are *not* already present. These installers will not overwrite later versions of the libraries.

# Index

## *Numerics*

## *A*

# S

## *T*

## *U*

## Z

*Application Programming*