IDL Tutorial

Programming in IDL

A Brief Tour of the IDL Language

Individuals who are new to IDL or have not been exposed to the software before may be interested in seeing some demonstrations of how the software can be utilized. One of the best ways to get a feel for the capabilities of IDL is to run the built-in demo system, which can be accessed by executing the command *DEMO* at the IDL> command prompt :

```
IDL> demo
```

Note: The IDL source code for these demonstration programs can be found in the "*examples/demo/demosrc/*" subfolder of the IDL installation.

The demonstration programs are launched by a category on the left-hand side, then double-clicking on the link in the lower-right hand corner.

Although IDL has a number of interactive tools for data input, analysis, and visualization, it is (in essence) a programming language. The primary mechanism for using IDL is by executing **statements**, either at the command line or within programs, which control the actions of IDL. IDL statements are case insensitive (and most are also space insensitive to an extent). For instance, each of the following are acceptable IDL statements (and perform the same exact operation) :

1. IDL> `PRINT,2*4`
   ```
   8
   ```
2. IDL> `print, 2 * 4`
   ```
   8
   ```
3. IDL> `Print, 2*4`
   ```
   8
   ```

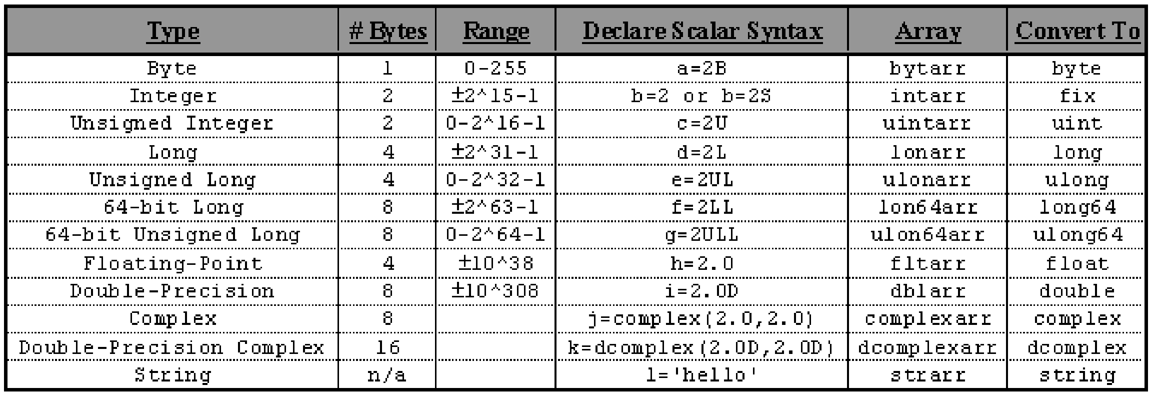
The *PRINT* routine prints the value of its argument into the IDL output log. Notice that the argument is evaluated before it is printed, resulting in the number (8) being output to the log window.

Note: IDL saves previously entered statements in a buffer, and these statements can be recalled to the command line with the UP-DOWN arrow keys on most keyboards while cursor focus is at the command line. The number of lines saved in the recall buffer can be changed in the Preferences for IDL.

In an IDL session, data (i.e. numbers and strings) is stored in what's known as **variables**. There are 3 basic types of IDL variables :

- **Scalar** (a single value)
- **Array** (from 1 to 8 dimensions – a **vector** is an array with only 1 dimension)
- **Structure** (an aggregate of various data types and other variables)

Any given variable in IDL also has a specific **data type**. There are 12 basic atomic data types in IDL (seven different types of integers, two floating-point types, two complex types, and a string) [Fig. 1-3]. 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 (i.e. the IDL language is **dynamically typed**).



Type	# Bytes	Range	Declare Scalar Syntax	Array	Convert To
Byte	1	0-255	a=2B	bytarr	byte
Integer	2	±2^15-1	b=2 or b=2S	intarr	fix
Unsigned Integer	2	0-2^16-1	c=2U	uintarr	uint
Long	4	±2^31-1	d=2L	lonarr	long
Unsigned Long	4	0-2^32-1	e=2UL	ulonarr	ulong
64-bit Long	8	±2^63-1	f=2LL	lon64arr	long64
64-bit Unsigned Long	8	0-2^64-1	g=2ULL	ulon64arr	ulong64
Floating-Point	4	±10^38	h=2.0	fltarr	float
Double-Precision	8	±10^308	i=2.0D	dblarr	double
Complex	8		j=complex(2.0,2.0)	complexarr	complex
Double-Precision Complex	16		k=dcomplex(2.0D,2.0D)	dcomplexarr	dcomplex
String	n/a		l='hello'	strarr	string

Figure 1-3: IDL data types

Variables do not have to be declared in IDL, and the data type of a variable can be determined by its usage. If a new variable is set equal to the sum of two integers, then the new variable will also be an integer. For example, start by declaring a scalar variable named "*a*" and set it equal to a value of (2) :

```
4. IDL> a = 2
```

By default, if a variable is set to a whole number it is assigned the 16-bit signed integer data type, as illustrated in the *Declare Scalar Syntax* column of Fig. 1-3. Next, declare a second scalar variable called "*b*" and set it equal to a floating-point value of (5) :

```
5. IDL> b = 5.0
```

IDL provides the ability to perform an arithmetic operation on these two variables and store the result in a new variable called "*c*" without having to first declare "*c*" :

```
6. IDL> c = a + b
```

The *HELP* routine is used to get information about the IDL session. In this case, use the *HELP* routine to obtain information on the three variables declared thus far :

```
7. IDL> help, a, b, c
   A               INT       =        2
   B               FLOAT     =       5.00000
   C               FLOAT     =       7.00000
```

Notice that when variables of different data types are combined in a single expression, the result has the data type that yields the highest precision.

So far all of the work performed has been with scalar variables, but the real power of IDL is in the fact that it's an **array-oriented** language. For example, declare a variable called "*array*" as an integer matrix with two dimensions of size 5 columns and 5 rows. Use the *INDGEN* function to set the value for each element of the array to its one-dimensional subscript (notice that IDL is a **row-major** language) :

```
8.  IDL> array = indgen (5, 5)
9.  IDL> help, array
    ARRAY           INT       = Array[5, 5]
10. IDL> print, array
           0       1       2       3       4
           5       6       7       8       9
          10      11      12      13      14
          15      16      17      18      19
          20      21      22      23      24
```

Since IDL is an array-oriented language, any operation that is applied to an array variable will automatically affect every element of the array without having to utilize FOR loops. For example, every element within the variable "*array*" can be multiplied by the scalar value (2) that is currently stored in the variable "*a*" with one simple statement :

```
11. IDL> array = array * a
12. IDL> print, array
           0       2       4       6       8
          10      12      14      16      18
          20      22      24      26      28
          30      32      34      36      38
          40      42      44      46      48
```

The ability to perform operations on only specific elements of an array is another power of the IDL language and is called **subscripting**. The square bracket characters "[" and "]" are used to perform subscripting in IDL. Since IDL is row-major, the appropriate way to subscript a 2-dimensional array is with the syntax [*column#*, *row#*]. It is also important to keep in mind that indexes for subscripting start at 0 instead of 1. For example, to print only the value found in the element in the 2nd column and 4th row of the variable "*array*" execute the statement :

```
13. IDL> print, array [1,3]
          32
```

So far we have executed two different routines from the IDL language library, *PRINT* and *HELP*. There are 3 basic types of routines that can be used by executing statements within the IDL language :

- **Procedures**
- **Functions**
- **Executive Commands**

A **procedure** is a routine that simply performs a well-defined task. In contrast, a **function** is a routine that performs a well-defined task and also returns a value to the specified variable once it is finished executing. **Executive commands** are used to control the execution of IDL programs. The statements executed to run the 3 different types of IDL routines differ in their calling sequence syntax :

Procedure: *IDL>* ***PROCEDURE,*** *argument*
Function: *IDL>* *result =* ***FUNCTION*** *(argument)*
Executive: *IDL>* ***.EXECUTIVE_COMMAND*** *-flags argument*

For example, execute the following procedure, function, and executive command statements at the IDL> command prompt (Note: "0" is the number zero) :

14. *Procedure:* IDL> **CALENDAR**, 1976
15. *Function:* IDL> time = **SYSTIME** (0)
16. *Executive:* IDL> .**COMPILE** arrow

In the example statements above, the *CALENDAR* procedure displays a simple calendar for the year 1976 in an IDL graphics window, the *SYSTIME* function returns the current time as a date/time string into the variable "*time*", and the *.COMPILE* executive command compiles the *ARROW* routine from the IDL distribution library and opens its source code file into the IDLDE document panel. There are hundreds of routines built into IDL, and most fall within one of these three categories.

In order to view the value returned by the *SYSTIME* function into the variable "*time*" the *HELP* procedure can be utilized :

17. IDL> `help, time`

The IDL output log should report that "*time*" is a variable of type string that is equal to the current date/time in the format "*DOW MON DD HH:MM:SS YEAR*".

There are also 2 different types of **parameters** that any given IDL routine can accept :

- **Arguments**
- **Keywords**

In the examples above, only arguments were specified. **Arguments** are used to *pass information* (e.g. data) to and/or from the IDL routine. Arguments are *positional* in that the order in which they are passed dictates what the IDL routine does with the information in the variable/value specified. Some or all of the arguments for any given routine may be optional.

In contrast, **keywords** are always optional and can be specified in any order. Keyword parameters are used to *control the behavior* of the IDL routine being executed, or to specify a *named variable* into which a result will be placed. For instance, to control the behavior of the *HELP* procedure so it returns information about the amount of dynamic memory (in bytes) currently in use by the IDL session, set the *MEMORY* keyword equal to one :

18. IDL> `help, memory=1`

The *MEMORY* keyword is a binary behavioral parameter that is either "on" or "off", and this is controlled by setting the keyword to a value of 1 or 0, respectively. A shortcut to *setting* a keyword "on" is available by preceding the variable with the forward slash "/" character. In addition, keywords can be abbreviated to the smallest string that uniquely identifies them for the routine in question. For example, the following is a shortcut to perform the same exact task as above :

```
19. IDL> help, /mem
```

Some keywords are used to return results from a routine. For example, the *OUTPUT* keyword to the *HELP* procedure can be used to place a string array containing the formatted output of the *HELP* command into a named variable called "*text*" :

```
20. IDL> help, /mem, output=text
21. IDL> help, text
    TEXT            STRING    = Array[1]
22. IDL> print, text
```

Finally, it is worth mentioning that you can view the IDL software documentation and automatically jump to the index location of a specific keyword by executing the question-mark character "?" followed by the keyword :

```
23. IDL> ?help
```

This will launch the IDL Online Help system and display the reference guide entry for the *HELP* procedure.

Introduction to IDL Programming

An IDL program consists of one or more IDL commands that are executed in a sequential fashion. The IDL software 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 compiled computer languages such as FORTRAN or C. Using IDL, tasks which require days or weeks of programming with traditional languages can be accomplished in hours. The user can explore data interactively using IDL commands and then create complete applications by writing IDL programs. Advantages of programming in IDL include :

- IDL is a complete, structured language that can be used both interactively and to create sophisticated algorithms and interactive 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 the user to observe the results of 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 the user 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.

- 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 the program 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".

There are several different ways in which a computer program can be created, and IDL supports the development of programs using a wide variety of methodologies. There are 5 primary types of IDL programs :

- Batch Files
- Main-Level Programs
- Named Programs (procedures & functions)
    - Object-Oriented Programs (creating IDL objects and their methods)
    - iTools System Programs

These different types of IDL programs are not necessarily completely distinct from one another, and each may or may not include components from IDL's widget toolkit in order to display a graphical user interface. For example, in order to create objects in IDL the programmer must actually write a series of named procedures or functions. Furthermore, iTools system programming is merely an extension of IDL's object-oriented methodology that focuses on customizing and extending the iTools.

IDL is inherently a **procedural language**, which means that the programmer specifies an explicit sequences of steps to follow to produce a result. A procedural program is written as a list of instructions telling the computer, step-by-step, what to do (e.g. open a file, read in data, perform processing, display result, etc.). Procedural programming is a method of computer programming based upon the concept of the unit and scope (the data viewing range of an executable code statement). It is possible for a procedural program to have multiple levels or scopes, with procedures defined inside other procedures. Each scope can contain variables which cannot be seen in outer scopes.

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. 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 program.

In the following exercise, a simple batch file is created and executed using the at symbol ("@") special character. The "@" symbol is either used as an include character within other programs or to signal that batch processing is to be performed. Executing the "@" symbol followed by the path to a batch file on disk will execute all of the statements within the batch file, one line at a time, in a sequential fashion.

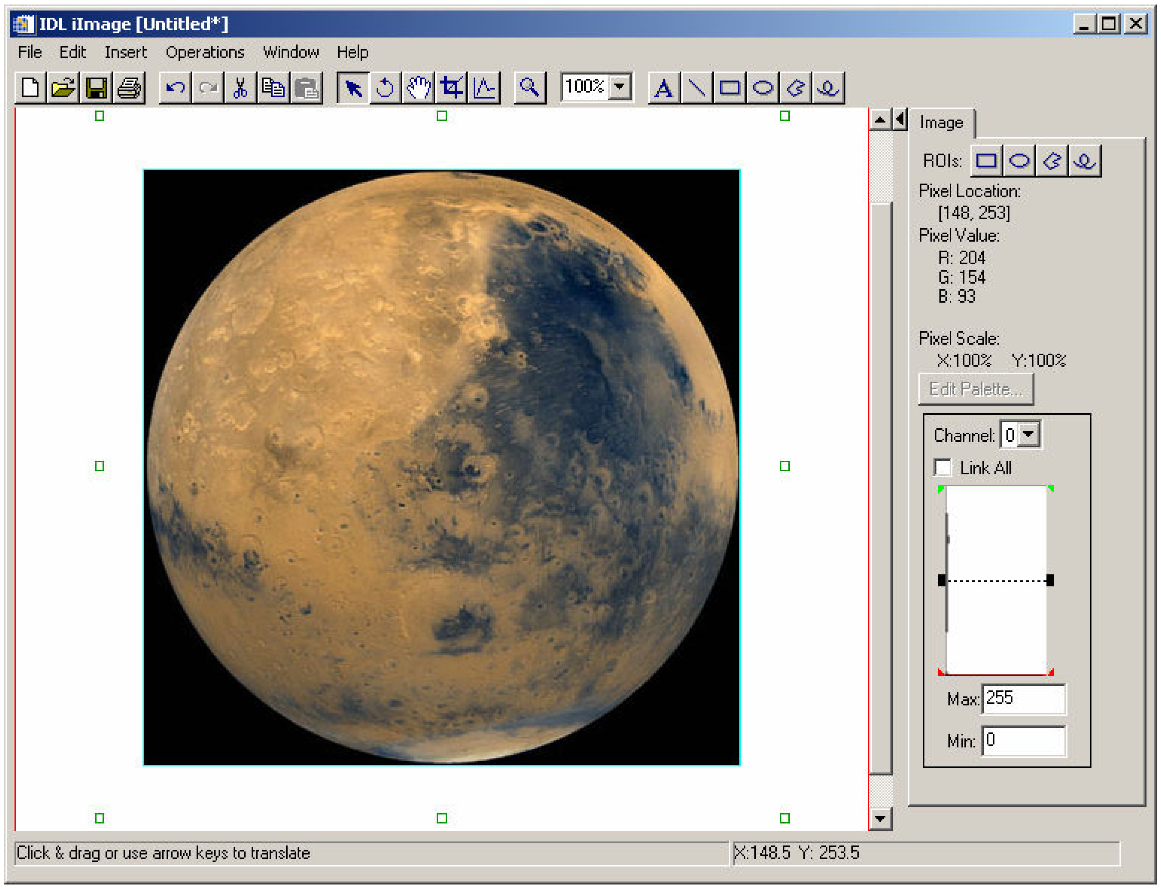
1. Start by selecting "*File > New > IDL Source File*" from the main IDL Development Environment menu system. This will open a new blank text editor window in the document panel of the IDLDE.
2. Within the text editor window, enter the following IDL code :

```
file = filepath('marsglobe.jpg', subdir=['examples', 'data'])
read_jpeg, file, image
iImage, image
```

3. Once these 3 lines of IDL code have been entered, select "*File > Save As…*" from the menu system.
4. Save the text to a new file named "*batch.pro*" located within the path for your IDL installation.
5. Finally, execute this batch file :

```
IDL> @batch
```

This will execute each of the 3 lines from the batch file in sequence as if they were executed individually at the IDL> command prompt. The resulting *iImage* visualization window should look similar to Fig. 1.



Pixel Location:
[148, 253]
Pixel Value:
R: 204
G: 154
B: 93

Pixel Scale:
X:100% Y:100%

Channel: 0
Max: 255
Min: 0

Click & drag or use arrow keys to translate
X:148.5 Y: 253.5

Figure 1: Result of executing batch file program

6. Once finished viewing the image, close the *IDL iImage* window.
7. Select "*File > Close*" in order to close the text editor window for this batch file.

Main-Level Programs

Main-level programs are entered at the IDL command line, and are useful when you have a few commands you want to run without creating a separate file to contain your commands. Main-level programs are not explicitly named; they consist of a series of statements that are not preceded by a procedure or 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 as the result of a *.RUN* executive command, it compiles it into the special program named *$MAIN$* and immediately executes it. Afterwards, it can be executed again by using the *.GO* executive command.

The following example creates and executes a small main-level program :

1. At the IDL> command prompt, enter the following :

```
IDL> a = 3
```

2. Next, execute the *.RUN* executive command. This changes the command prompt from "IDL>" to "-" :

```
IDL> .RUN
```

3. Enter the following 3 statements at the "-" prompt :

```
a = a ^ 2
PRINT, a
END
```

4. This creates a main-level program, which automatically compiles and executes, resulting in the following output :

```
% Compiled module: $MAIN$.
       9
```

5. This main-level program can be run again by executing the *.GO* executive command :

```
IDL> .GO
```

6. The result of executing this program a second time is cumulative, resulting in the following output :

```
      81
```

Named Programs (Procedures & Functions)

Named programs are modules that are given explicit names so they can be called from other programs as well as executed at the IDL command line. Named programs are usually stored in ASCII text files on disk and are given a "*.pro*" extension by convention. There are two different types of named programs in IDL; procedures and functions. The concept of procedures and functions should be familiar since almost all of the statements executed in this tutorial thus far have involved either a procedure or a function (or a combination of both).

Procedures and functions are self-contained modules that break large tasks into smaller, more manageable ones. A **procedure** is a self contained sequence of IDL statements with an unique name that performs a well defined task. 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. Consequently, all functions must contain a call to the *RETURN* procedure with a specific value (scalar, string, array, structure, etc.) as the argument which is returned to the calling program unit.

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 *SURFACE*) 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 compile the program before it can be executed. There are 3 ways in which a procedure or function can be compiled :

- Automatically : If the ASCII text source code file for the program has a filename that is identical to the name of the main program module, the filename ends with a "*.pro*" extension, and this file is found within IDL's path, then the program will be automatically compiled when it is executed. For reference, IDL's path is one of the Preferences found within the IDL Development Environment, and it is also stored in an internal system variable named *!PATH*.
- Interactively : If the source code file for the program is currently open within the text editor of the IDL Development Environment, the user can either select "*Run > Compile*" from the menu system or press the yellow compile button on the toolbar in order to compile the program.
- Manually : The user can explicitly compile a program by executing the *.COMPILE*, *.RUN*, or *.RNEW* executive commands with the appropriate path to the source code file on disk. If the source code file is found within IDL's path or the current working directory, then just the name of the file itself is needed. If the user is working within the IDL Development Environment, the source code will automatically be opened into the text editor window.

In the following exercise, a simple procedure program named "*muscle_view*" will be created, compiled, and executed. This program will open the example data file "*muscle.jpg*" included with the IDL installation, read the image data into IDL, and displays the image in a *FOR* loop that cycles through all of IDL's 41 predefined colortables using Direct Graphics.

1. Start by selecting "*File > New > IDL Source File*" from the main IDL Development Environment menu system. This will open a new blank text editor window in the document panel of the IDLDE.
2. The first step in writing a procedure is to create the **definition statement**, and this is accomplished using the *PRO* statement in IDL. In this case, the name given to the procedure will be "*muscle_view*", so start by typing the following text in the blank text editor window :

```
PRO muscle_view
```

At this point, the name of the current text document is probably [Untitled1*] and it is not being saved to the harddrive, so it may be appropriate to save this text to an IDL source code file on disk.

3. From the main IDLDE menu system select "*File > Save As*".
4. Save the text to a new file named "*muscle_view.pro*" located within the path for your IDL installation.

Once this is accomplished, the title bar across the top of the IDL Development Environment should be labeled with "[muscle_view.pro]". Whenever a change is made to the file an asterisk "*" character will be added to the end of the filename, alerting the user that the file contains modifications that have not yet been saved to the file on disk. Changes to the source code file can be saved by selecting "*File > Save*" from the IDLDE menu or by pressing the 💾 button on the IDLDE toolbar.

5. The first step within the program that needs to be entered on the next line of the text editor is the specification of the path to the file that is going to be opened :

```
file = filepath('muscle.jpg', subdir=['examples','data'])
```

6. When writing a program it is always a good idea to include as much error checking as possible. In this case it may be a good idea to make sure that the specified file is a valid JPEG image file and obtain some information about it using the *QUERY_JPEG* function before proceeding :

```
result = query_jpeg(file, info)
```

7. The *QUERY_JPEG* function returns a value of "*1*" into the result if the query was successful and the file appears to be a valid JPEG format image file, otherwise it returns "*0*" upon failure. Consequently, it may be a good idea to terminate the execution of this program at this point using the *RETURN* procedure if the file is not determined to be a valid JPEG image file :

```
if result eq 0 then return
```

8. Now the JPEG image file data can be read into this IDL program unit using the *READ_JPEG* procedure :

```
read_jpeg, file, image
```

9. An IDL graphics window needs to be created to display this image, and the size of this window can be made to match the size of the image using the "*info*" structure returned by the earlier call to *QUERY_JPEG* :

   ```
   window, xsize=info.dimensions[0], ysize=info.dimensions[1]
   ```

10. Next, the use of color decomposition must be disabled in order to use IDL's built-in colortables on 24-bit displays :

    ```
    device, decomposed=0
    ```

Now the program is ready to cycle through the colortables and display the image in the Direct Graphics window. In order to accomplish this task, the programmer must make use of the *FOR* control statement. The syntax for declaring a *FOR* loop within the IDL language is :

```
FOR variable = init, limit DO BEGIN
  statements
ENDFOR
```

11. In order to loop through each of IDL's predefined colortables and display the image data insert the following lines of text :

    ```
    for i = 0, 40 do begin
      loadct, i
      tvscl, image
    endfor
    ```

This will cycle through the variable "*i*" set to values 0→40, load the colortable index for the current *FOR* loop iteration value of "*i*" using *LOADCT*, display the image using *TVSCL*, increment the variable "*i*" by a value of 1, and start the next iteration.

12. Once the *FOR* loop has executed, the graphics window that was created can be cleaned up and destroyed by calling the *WDELETE* procedure :

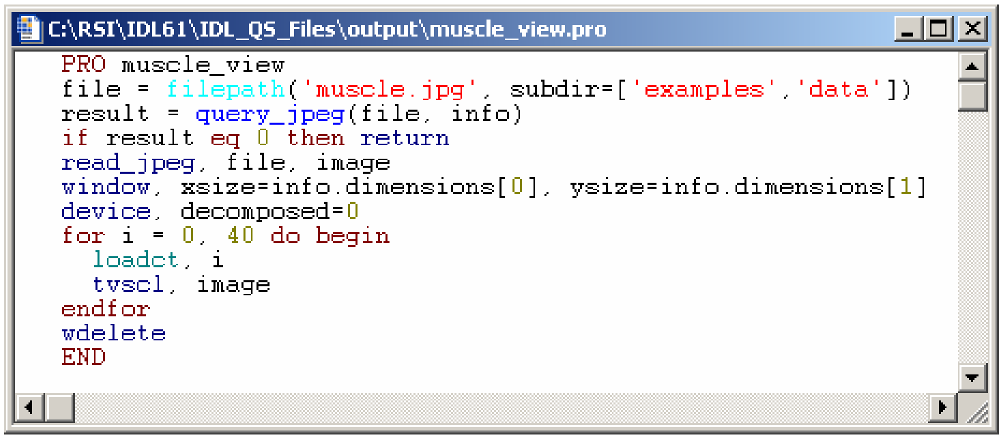
    ```
    wdelete
    ```

13. Finally, the end of the program must be defined by inserting the *END* statement :

    ```
    END
    ```

14. Once all of this text has been entered make sure to select "*File > Save*" or hit the 💾 button on the IDLDE toolbar.

The final program within the text editor window should look similar to Fig. 2.



```
PRO muscle_view
file = filepath('muscle.jpg', subdir=['examples','data'])
result = query_jpeg(file, info)
if result eq 0 then return
read_jpeg, file, image
window, xsize=info.dimensions[0], ysize=info.dimensions[1]
device, decomposed=0
for i = 0, 40 do begin
  loadct, i
  tvscl, image
endfor
wdelete
END
```

Figure 2: Completed source code for the "muscle_view" procedure

The first step in running this IDL program is to compile the procedure so it can be executed within the IDL process.

15. In order to compile the procedure, simply select "*Run > Compile muscle_view.pro"* from the main IDLDE menu system or hit the yellow compile button on the toolbar. This will automatically call the executive command *.COMPILE* and the output log should read :

```
% Compiled module: MUSCLE_VIEW.
```

If there are any compilation errors an informational message will appear in the output log and a red circular dot will be placed to the left of the line where the error occurs. Any compilation errors must be resolved before the program can be executed.

16. Once the program is successfully compiled it can be executed by selecting "*Run > Run muscle_view"* from the menu, hitting the blue run button on the toolbar, or simply executing "*muscle_view*" at the IDL> prompt :

```
IDL> muscle_view
```

During execution of the program the "*muscle.jpg*" image will be displayed using all 41 of IDL's built-in colortables [Fig. 3]. While the program is executing the *LOADCT* procedure will output the currently loaded colortable to the output log when it's being called within the *FOR* loop. The total time it takes for this program to execute should be very short, which is a testament to the rapid image display capabilities of the Direct Graphics system and the overall speed of the IDL language.

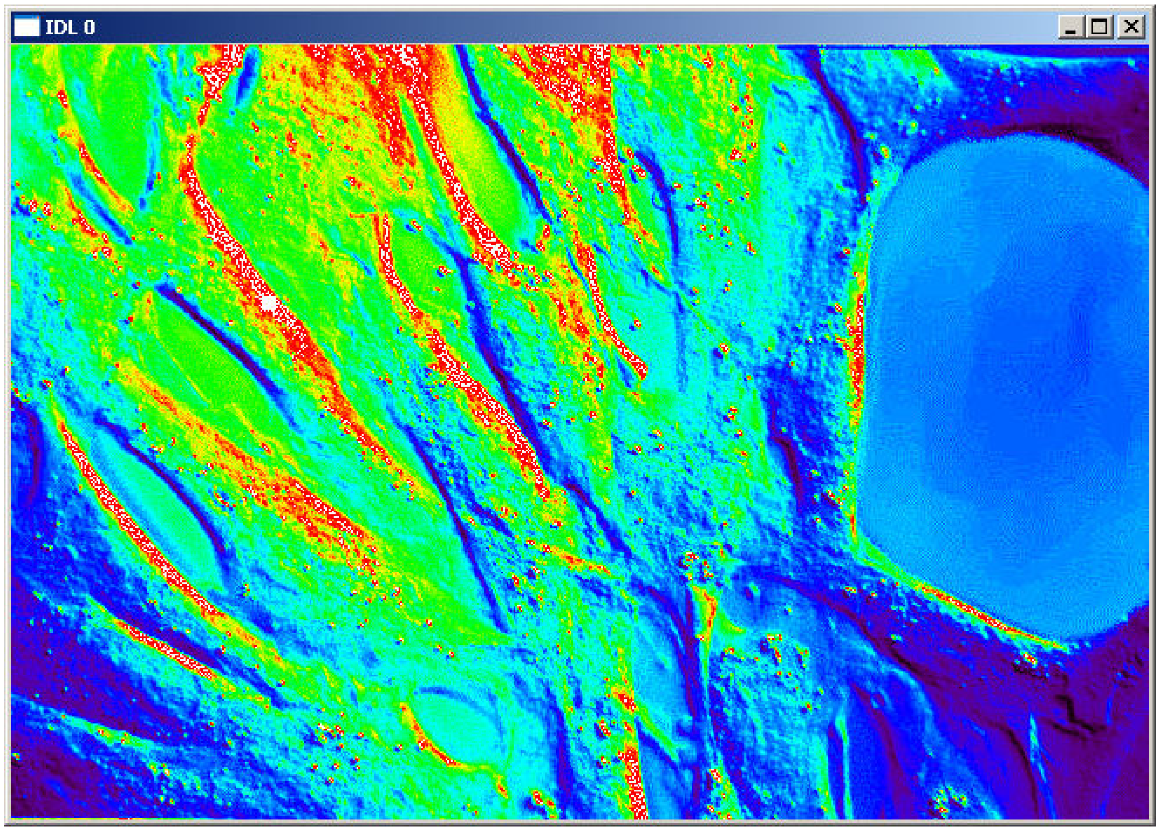


Figure 3: Image display during execution of the "muscle_view" program

17. Select "*File > Close*" in order to close the text editor window for this procedure.

In the next exercise a simple function will be created that computes the area of a circle given its radius. Functions are slightly different from procedures because they must return a specific value to the calling module. Consequently, functions are particularly useful when performing data processing and analysis. In addition, as previous exercises within this tutorial have illustrated, the calling syntax for executing functions is different than procedures.

18. Start by selecting "*File > New > IDL Source File*" from the main IDL Development Environment menu system. This will open a new blank text editor window in the document panel of the IDLDE.

Once again, the first step in writing a function is to create the **definition statement**, and this is accomplished using the *FUNCTION* statement in IDL. In this case, the function will need to accept an **argument** in order to allow passing of the input radius value into the program. Furthermore, the user may wish to specify whether they want the calculation performed using single-precision or double-precision floating-point arithmetic. This can be accomplished by specifying a **keyword** that the user can utilize in order to control the behavior of the function.

19. The name given to the function will be "*circle_area*", the input argument should be named "*radius*", and a keyword named "*dbl*" can be declared so the

user has control over the precision. This can be accomplished with the following definition statement :

```
FUNCTION circle_area, radius, dbl=dbl
```

Once again, it is a good idea to save this text to an IDL source code file on disk :

20. From the main IDLDE menu system select "*File > Save As*".
21. Save the text to a new file named "*circle_area.pro*" located within the path for your IDL installation.
22. The *KEYWORD_SET* function built into the IDL language library is useful for determining the status of a keyword variable and whether or not it was "*set*" by the user. In this case, a variable named "*dbl*" exists within the function and either has a value of *1* if it was set, or *0* if it was left un-set. The *KEYWORD_SET* function can be used in conjunction with an IF … THEN … ELSE code block to perform the necessary calculation using the appropriate IDL system variable for the value of ∏ :

```
if keyword_set(dbl) then begin
  area = !DPI * radius ^ 2
endif else begin
  area = !PI * radius ^ 2
endelse
```

23. Once the area has been computed using the appropriate precision, the "*area*" variable can be returned to the calling program module :

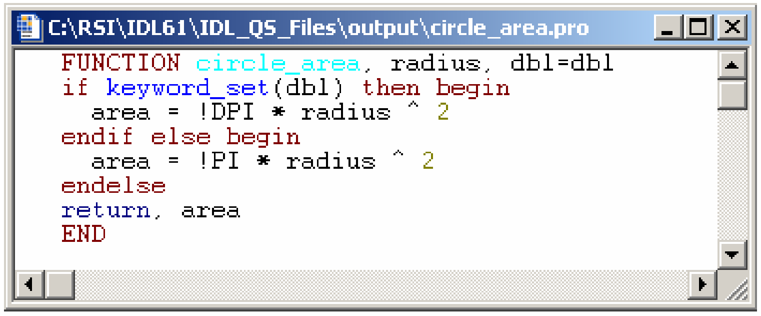
```
return, area
```

24. Finally, the end of the program must be defined by inserting the *END* statement :

```
END
```

25. Once all of this text has been entered make sure to select "*File > Save*" or hit the button on the IDLDE toolbar.

The final program within the text editor window should look similar to Fig. 4.


C:\RSI\IDL61\IDL_QS_Files\output\circle_area.pro

```
FUNCTION circle_area, radius, dbl=dbl
if keyword_set(dbl) then begin
  area = !DPI * radius ^ 2
endif else begin
  area = !PI * radius ^ 2
endelse
return, area
END
```

Figure 4: Completed source code for the "circle_area" function

Before the "*circle_area*" function can be executed it must be compiled.

26. In order to compile the function, simply select "*Run > Compile circle_area.pro"* from the main IDLDE menu system or hit the yellow compile button on the toolbar. This will automatically call the executive command *.COMPILE* and the output log should read :

```
% Compiled module: CIRCLE_AREA.
```

If there are any compilation errors an informational message will appear in the output log and a red circular dot will be placed to the left of the line where the error occurs. Any compilation errors must be resolved before the program can be executed.

Once the program is successfully compiled it can be executed by calling it with the appropriate function syntax :

27. IDL> `area = CIRCLE_AREA (7)`

This statement calls the "*circle_area*" function and specifies an input radius value of seven. The named variable "*area*" contains the result of this calculation that is returned from the function :

28. IDL> `HELP, area`
```
AREA            FLOAT     =       153.938
```

In addition, the "*circle_area*" function can be called with the "dbl" keyword set in order to compute the area in double-precision :

29. IDL> `area = CIRCLE_AREA (7, /DBL)`
30. IDL> `HELP, area`
```
AREA            DOUBLE    =        153.93804
```

The "*area*" variable that is returned from the function is now double-precision.

31. Once finished experimenting with the "*circle_area*" function, select "*File > Close*" in order to close the text editor window.

Object-Oriented Programs

Traditional programming techniques make a strong distinction between routines written in the programming language (procedures and functions in the case of IDL) and the data to be acted upon by these routines. In contrast, object-oriented programming removes this distinction by encapsulating both data and functionality (i.e. routines) into a single entity known as **objects**.

IDL objects may contain data with various data types or organizational arrangement. The routines (i.e. functionality) within an object that act upon this data are called **methods**. Object methods are merely IDL procedures and functions that have special names and are called in a special way. In the IDL programming environment, object data are protected from the rest of the program and are only accessible through the object methods (i.e. IDL object data is always private).

IDL's object system provides support for the following concepts and mechanisms:

- **Classes and Instances** : IDL objects are created as instances of a class, which is defined in the form of an IDL structure. The name of the structure is also the class name for the object. Objects can only be created by calling the *OBJ_NEW* (or *OBJARR*) function with the name of the class structure as the argument, and can only be accessed via the returned object reference.
- **Encapsulation** : Encapsulation is the ability to combine data and the routines that affect the data (known as methods) into a single object.
- **Methods** : Methods are the routines (procedures and functions) that perform specific operations within the object. Method routines are identified as belonging to an object class via a routine naming convention *ClassName::MethodName*.
- **Polymorphism** : Polymorphism is the ability to create multiple object types that support the same operations.
- **Inheritance** : Inheritance is the ability of an object class to inherit the behavior of other object classes. This means that when writing a new object class that is very much like an existing object class, you need only program the functions that are different from those in the inherited class.
- **Persistence** : Persistence is the ability of objects to remain in existence in memory after they have been created, allowing you to alter their behavior or appearance after their creation. IDL objects persist until you explicitly destroy them, or until the end of the IDL session.
- **Object Lifecycle** : The life of an object can be broken into three phases: creation, use, and destruction. Object references are created using one of two lifecycle routines: *OBJ_NEW* or *OBJARR*. The object is used by executing its methods. Finally, the object is destroyed using the *OBJ_DESTROY* procedure.

Objects are implemented in IDL as extensions of data structures. In other words, a data structure and one or more IDL routines are combined to form an IDL object **class**. Object classes are essentially the blueprints used to define individual objects. There are 2 basic steps to creating an object class in IDL :

- Define the object class data structure by creating a named structure. The name of this structure must be the same as the desired name for the object class. For example, the following would be an appropriate structure definition for an object class named *SHOW3OBJ* :

  ```
  namedStructure = {SHOW3OBJ, data:ptr_new()}
  ```

- Define the methods of the object class using the *CLASSNAME::method* declaration syntax. For example, a method named "*Display*" could be defined for the *SHOW3OBJ* class using the following syntax :

  ```
  PRO SHOW3OBJ::Display
  if ptr_valid(self.data) then show3, *self.data
  END
  ```

Once the object class has been defined an instance of the object can be created using the *OBJ_NEW* function :

```
oShow3 = OBJ_NEW ('SHOW3OBJ', dist(100))
```

The *Display* method to this object can then be executed using IDL's method invocation operator ("->"), which is a hyphen followed by a greater-than sign :

```
oShow3 -> display
```

In this example the *Display* method is a procedure, but methods can also be functions. If an object method is defined as a function then its calling sequence will be :

```
result = object -> functionMethod ()
```

Once an object has data assigned to it, the methods automatically have access to this data in a structure called "*self*" that has the same layout as the original object class data structure. Consequently, there is no need to pass the data via arguments and/or keywords in the call to an object method. In addition, the *self* structure can be used to call other methods within an object. For instance, if there was a *LOADCOLOR* method to the *SHOW3OBJ* object class it could directly call the *DISPLAY* method using the *self* structure :

```
PRO SHOW3OBJ::Loadcolor, index
if n_params() NE 1 then return
if !D.N_COLORS GT 256 then device, decomposed=0
loadct, index
self -> display
END
```

When an instance of an object is created from a class using the *OBJ_NEW* function an **object reference** is returned that points to an object **heap variable** [Fig. 5]. A heap variable is an area of common memory allocated for a specific use and accessed by way of one or more reference variables. These reference variables are the means through which the object is referenced in IDL.

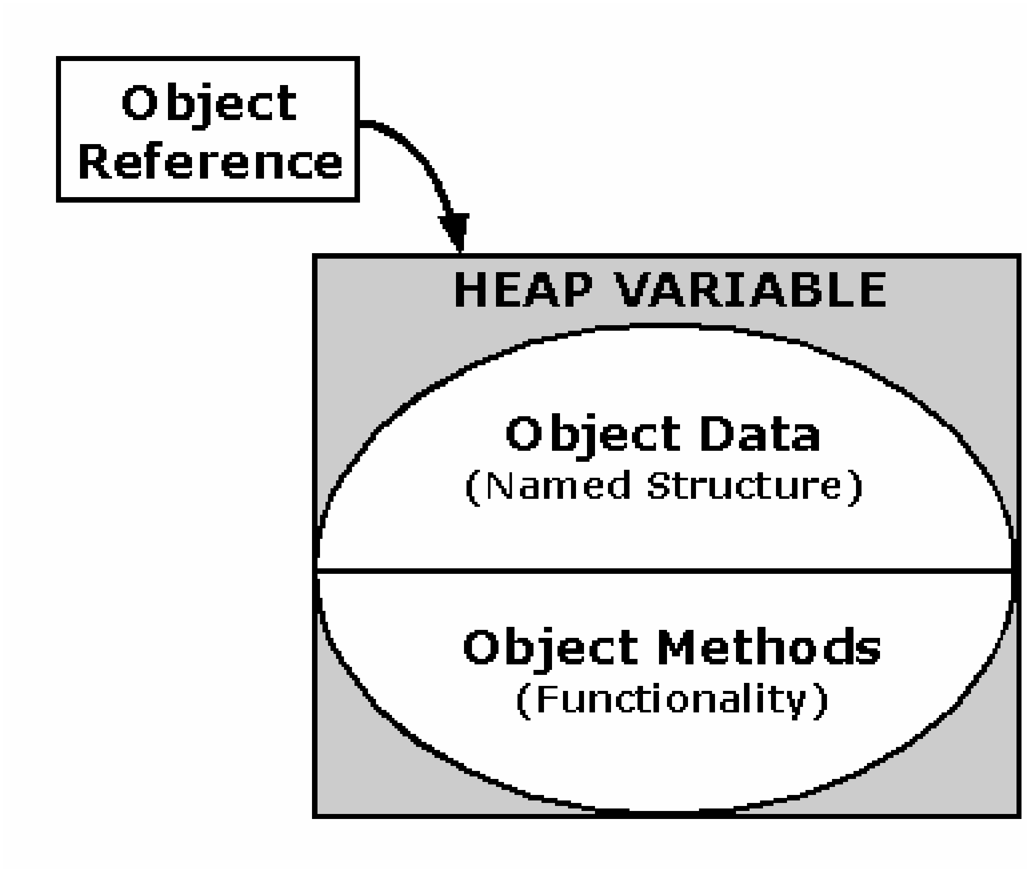


Figure 5: An object heap variable with an object reference. The heap variable encapsulates both the data and methods for the object.

Consequently, in the hypothetical example above, the variable *oShow3* is not actually the object itself but a reference to the object contained in heap memory. Consequently, the programmer must take care when deleting objects since the destruction of the object reference variable will not cleanup the heap memory occupied by the object. Object heap variables persist in memory until explicitly destroyed by the *OBJ_DESTROY* procedure:

```
OBJ_DESTROY, oShow3
```

Once the *OBJ_DESTROY* procedure is called the object heap variable for this object no longer exists, but the object reference variable *oShow3* still exists and points to a non-existent piece of memory. This is known as a **dangling reference**. Consequently, it is always a good idea to also destroy all object reference variables when destroying an object by either using the *DELVAR* procedure (interactive IDL only) or the *TEMPORARY* function:

```
OBJ_DESTROY, TEMPORARY(oShow3)
```

In order to create an object class in IDL there are a set of programming rules that must be followed when writing the source code. Selecting a name for a custom object class is a very important consideration because it is important not to use the same name as a class already built into IDL. Fortunately, most object classes built into IDL have names that start with the string "*IDL*", so it should be easy to derive a name that does not conflict with one of the built-in classes.

When creating a custom object class in IDL a special naming convention must be used for the source code file name and the procedure/function names within that source code file. As mentioned before, the general form for the procedure/function declaration statements within the object class source code is "*CLASSNAME::method*". In addition, there is also a naming convention for the source code file itself that follows the general form "*CLASSNAME__method*" (Note: "__" is two underscores).

The information necessary for the object must be defined in the object class data. When defining the data of an object, the programmer must determine in advance all of the various data elements that might be necessary to complete all of the functionality for the object. As mentioned earlier, the object class data is defined and stored in an IDL named structure that has the same name as the class itself. This is the first step in creating an object class and it occurs within a special procedural method called "*DEFINE*". Consequently, in order for IDL to create an instance of an object with *OBJ_NEW* an IDL source code file for this object with the name "*CLASSNAME__define.pro*" must be found in the *!PATH* (or manually compiled within the current session of IDL). The define method is also the only method which uses the two underscores ("__") in the procedural declaration statement as opposed to the two colons ("::"). For example, the source code to define the *SHOW3OBJ* object class above would look like:

```
PRO SHOW3OBJ__define
namedStructure={SHOW3OBJ, data:ptr_new()}
END
```

This source code needs to be saved in an ASCII text file called "*show3obj__define.pro*" in order for the object class to be used. The only purpose of the *__define* procedure is to create the named structure that will contain the data for the object class.

Once the data has been defined for an object class, the next step is to create the methods that will define the functionality. These methods can be stored in separate "*CLASSNAME__method.pro*" ASCII source code files, or they can all be stored in the main "*CLASSNAME__define.pro*" file as long as the *__define* procedure is the last module that occurs within the file.

There are also 2 other methods that must be included with every object class: *INIT* and *CLEANUP*. The 3 main object methods (*DEFINE, INIT, CLEANUP*) must be defined and compiled within IDL before an instance of the object can be created with *OBJ_NEW*. The *INIT* method is an IDL function that is called by *OBJ_NEW* when the object is created. The job of the *INIT* method is to take any arguments and keywords passed from the user and perform any initialization necessary for the object class. The process of initialization usually involves filling-in the members of the object data structure with the data passed by the user in the call to *OBJ_NEW*. Finally, the *INIT* method should return *1* if successful in initializing the object and *0* if the initialization failed. In contrast, the *CLEANUP* method is called when the object reference is destroyed (when *OBJ_DESTROY* is executed)) and should clean up any pointers, objects, or other miscellaneous data in an effort to avoid memory leakage.

The following exercise completes the definition of the *SHOW3OBJ* object class that has been discussed herein by writing its IDL source code.

1. Start by selecting "*File > New > IDL Source File*" from the main IDL Development Environment menu system. This will open a new blank text editor window in the document panel of the IDLDE.
2. Immediately select "*File > Save As"* and save this to a file called "*show3obj__define.pro*" located within the path for your IDL installation.
3. Type out the source code for this object class in the IDLDE text editor. There will be a total of 5 methods defined for the *SHOW3OBJ* object class: *DEFINE*, *INIT*, *CLEANUP*, *LOADCOLOR*, and *DISPLAY*. Make sure to remember the rule of placing the *__define* procedure at the bottom of this file :

```
FUNCTION SHOW3OBJ::Init, data
self.data = ptr_new(data)
if !d.n_colors gt 256 then device, decomposed=0
loadct, 0, /silent
return, 1
END

PRO SHOW3OBJ::Cleanup
ptr_free, self.data
print, 'SHOW3OBJ object successfully destroyed.'
END

PRO SHOW3OBJ::Display
if ptr_valid(self.data) then show3, *self.data
END

PRO SHOW3OBJ::Loadcolor, index
loadct, index
self -> display
END

PRO SHOW3OBJ__define
namedStructure = {SHOW3OBJ, data:ptr_new()}
END
```

4. Once all of this source code has been entered into the text editor, save the file and then compile the object class by pressing the yellow compile button. This should report the following in the IDL Output Log :

```
IDL> .COMPILE show3obj__define.pro
% Compiled module: SHOW3OBJ::INIT.
% Compiled module: SHOW3OBJ::CLEANUP.
% Compiled module: SHOW3OBJ::DISPLAY.
% Compiled module: SHOW3OBJ::LOADCOLOR.
% Compiled module: SHOW3OBJ__DEFINE.
```

Now that the object class is compiled within the current session of IDL it is ready to be used. Initialize an instance of the *SHOW3OBJ* object class with data created by the *DIST* function :

5. IDL> `oShow3 = OBJ_NEW ('SHOW3OBJ', DIST (100) )`

Next, call the *Display* method for the object :

6. IDL> `oShow3 -> Display`

The resulting *IDL 0* visualization window should contain a composite 3-D visualization including an image, wire-mesh surface, and contour plot produced by IDL's built-in *SHOW3* routine.

The *Loadcolor* method can also be executed to load IDL's Rainbow colortable and automatically redisplay the graphic:

7. IDL> `oShow3 -> Loadcolor, 13`

The resulting visualization window should look similar to Fig. 6 :

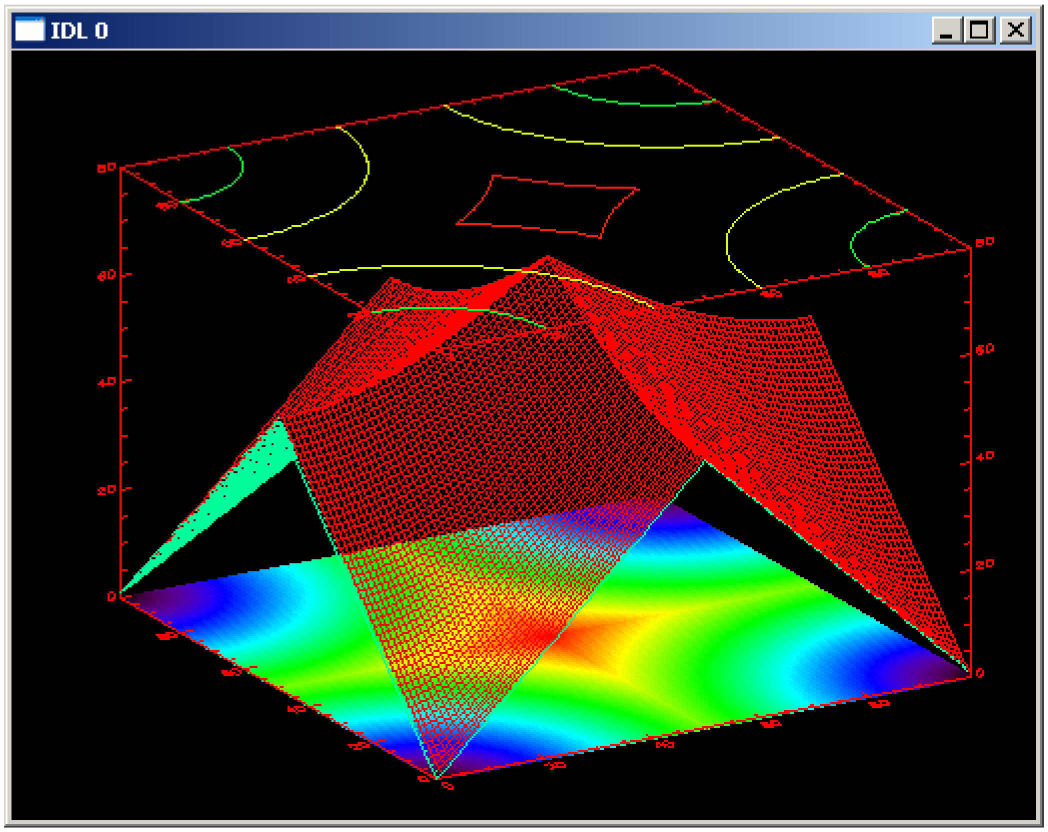


Figure 6: Result of invoking the "Loadcolor" method of the "SHOW3OBJ" object class

Once the user is finished working with the object it is important to destroy the heap variable and object reference:

8. IDL> `OBJ_DESTROY, TEMPORARY (oShow3)`
   ```
   SHOW3OBJ object successfully destroyed.
   ```
9. Once finished experimenting with the "*SHOW3OBJ*" object, select "*File > Close*" in order to close the text editor window.

iTools System Programs

iTools programming is a special form of object-oriented programming that is used to control, customize, and extend the tools built into the iTools system. The IDL Intelligent Tools, or *iTools*, are applications written in IDL to perform a variety of data analysis and visualization tasks. iTools share a common underlying application framework, presenting a full-featured, customizable, application-like user interface with menus, toolbars, and other graphical features.

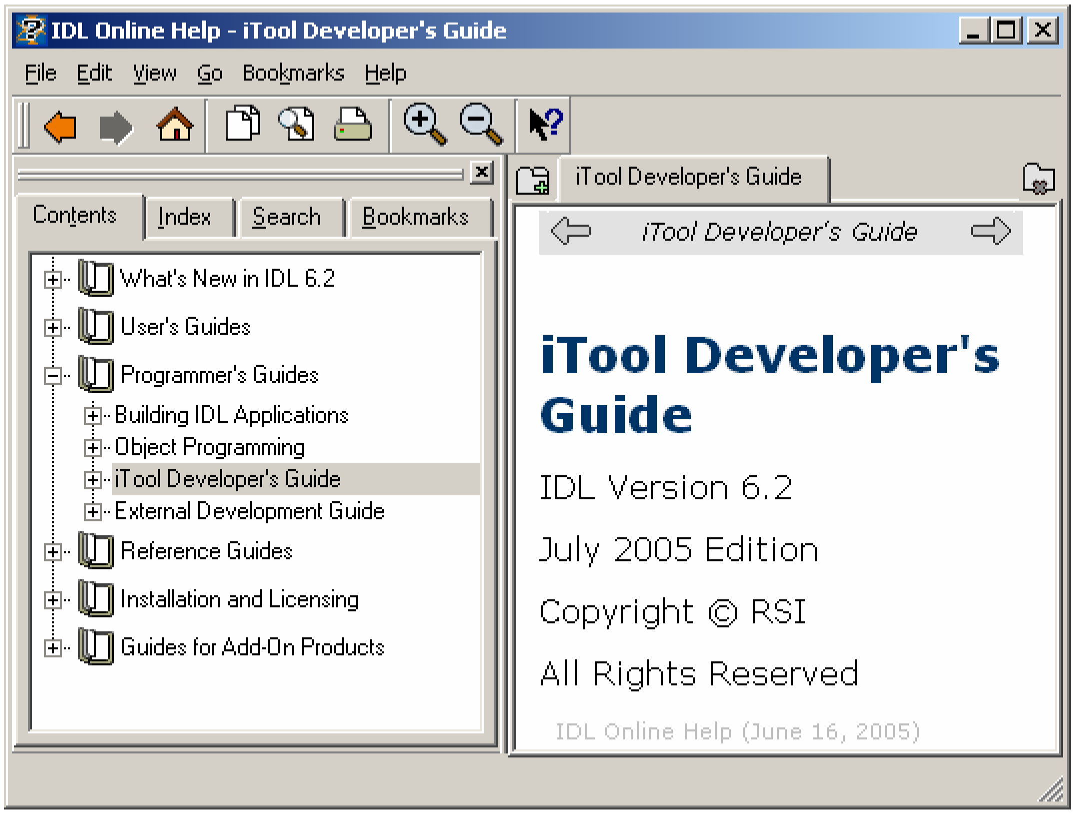
However, iTools are much more than just a set of pre-written IDL programs. Behind the iTool system lies the IDL Intelligent Tools Component Framework — a set of object class files and associated utilities designed to allow you to easily extend the supplied toolset or create entirely new tools of your own. The iTools component framework is a set of object class definitions written in the IDL language. It is designed to facilitate the development of sophisticated visualization tools by providing a set of pre-built components that provide standard features including :

- Creation of visualization graphics
- Mouse manipulations of visualization graphics
- Annotations
- Management of visualization and application properties
- Undo/redo capabilities
- Data import and export
- Printing
- Data filtering and manipulation
- Interface element event handling

In addition, the iTools component framework makes it easy to extend the system with components of your own creation, allowing you to design a tool to manipulate and display your data in any way you choose. Programming in the iTools system allows the user to create their own :

- iTool
- Visualization
- Operation
- Manipulator
- File Reader
- File Writer
- Graphical User Interface

A discussion of programming within the iTools system is beyond the scope of this tutorial. For more information please consult the *iTool Developer's Guide* documentation manual included with the IDL online help system [Fig. 7].


IDL Online Help - iTool Developer's Guide
File Edit View Go Bookmarks Help
Contents Index Search Bookmarks
What's New in IDL 6.2
User's Guides
Programmer's Guides
Building IDL Applications
Object Programming
iTool Developer's Guide
External Development Guide
Reference Guides
Installation and Licensing
Guides for Add-On Products
iTool Developer's Guide

iTool Developer's Guide

IDL Version 6.2

July 2005 Edition

Copyright © RSI

All Rights Reserved

IDL Online Help (June 16, 2005)

Figure 7: The iTool Developer's Guide documentation manual

Creating Graphical User Interfaces

IDL allows you to construct and manipulate graphical user interfaces using **widgets**. Widgets (or *controls*, in the terminology of some development environments) are simple graphical objects such as pushbuttons or sliders that allow user interaction via a pointing device (usually a mouse) and a keyboard. This style of graphical user interaction offers many significant advantages over traditional command-line based systems.

IDL widgets are significantly easier to use than other alternatives, such as writing a C language program using the native window system graphical interface toolkit directly. IDL handles much of the low-level work involved in using such toolkits. The interpretive nature of IDL makes it easy to prototype potential user interfaces. In addition to the user interface, the author of a program written in a traditional compiled language also must implement any computational and graphical code required by the program. IDL widget programs can draw on the full computational and graphical abilities of IDL to supply these components.

The style of widgets IDL creates depends on the windowing system supported by your host computer. Unix hosts use Motif widgets, while Microsoft Windows systems use the native Windows toolkit. Although the different toolkits produce applications with a slightly different look and feel, most properly-written widget applications work on all systems without change.

IDL graphical user interfaces are constructed by combining widgets in a treelike **hierarchy**. Each widget has one parent widget and zero or more child widgets. There is one exception: the topmost widget in the hierarchy (called a **top-level base**) is always a base widget and has no parent.

Programs that use widgets are **event driven**. In an event driven system, the program creates an interface and then waits for messages (events) to be sent to it from the window system. Events are generated in response to user manipulation, such as pressing a button or moving a slider. The program responds to events by carrying out the action or computation specified by the programmer, and then waiting for the next event. Because of widget applications' event-driven nature, creating applications that use widgets is fundamentally different from creating non-widget programs.

The following exercise creates a very simple graphical user interface application with the appropriate event handling sub-program. This program displays a simple GUI with a button labeled "*Display Image*" that can be pressed by the user in order to generate an event that display an image from an example file included with IDL.

1. Start by selecting "*File > New > IDL Source File*" from the main IDL Development Environment menu system. This will open a new blank text editor window in the document panel of the IDLDE.
2. Immediately select "*File > Save As"* from the menu system and save this document to a file called "*simple_gui.pro*" located within the path for your IDL installation.
3. Type out the source code for this widget program into the IDLDE text editor :

```
PRO simple_gui_event, event
widget_control, event.top, get_uvalue=state
tv, state.image, true=1
END

PRO simple_gui
tlb = widget_base(/row, title='Simple GUI')
  subBase=widget_base(tlb)
    button = widget_button(subBase, value='Display Image')
  draw = widget_draw(tlb, xsize=512, ysize=512)
widget_control, tlb, /realize
file = filepath('elev_t.jpg', subdir=['examples', 'data'])
read_jpeg, file, image
device, decomposed=1
state = {image:image}
widget_control, tlb, set_uvalue=state
xmanager, 'simple_gui', tlb
END
```

This program contains 2 modules; the main GUI creation procedure named "*simple_gui*", and the event handler procedure called "*simple_gui_event*". Within the "*simple_gui*" procedure the layout of the widget hierarchy is defined, the GUI is realized (i.e. displayed) to the screen, the image is read-in from a JPEG file on disk and stored in a state structure, and the *XMANAGER* routine is called in order to manage the event callback distribution. The "*simple_gui_event*" procedure is called when the user presses the "*Display Image*" button, and it obtains the state structure for the application and displays the image using the *TV* routine.

4. Once the source code for the program has been entered, save the changes by pressing the button on the IDLDE toolbar.
5. Compile the program by pressing the yellow compile button.
6. Finally, run the program by pressing the blue run button.
7. Once the GUI for the program is displayed, use the mouse to press the "*Display Image*" button [Fig. 8].

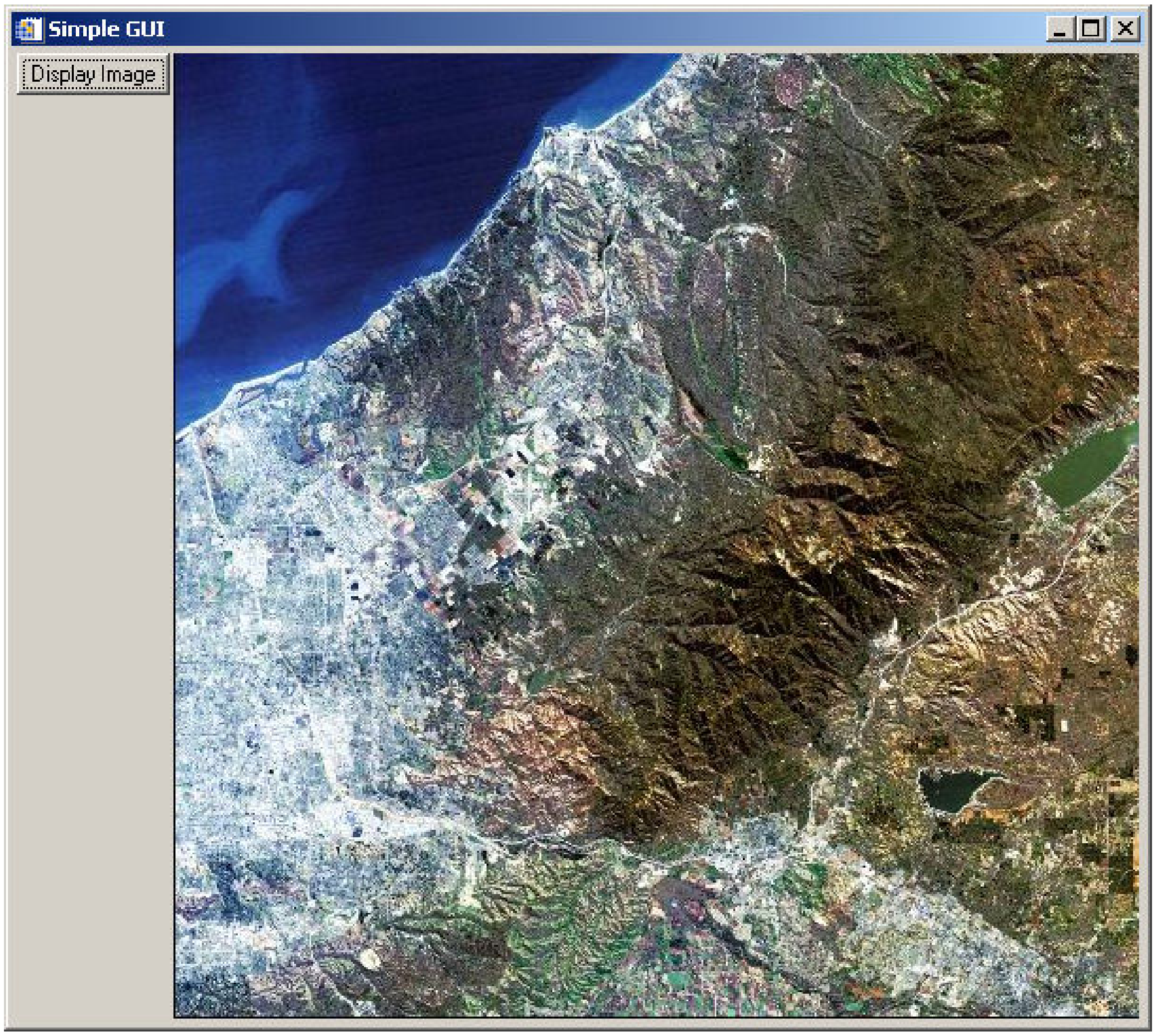


Figure 8: Execution of the "simple_gui" widget program

8. Once finished viewing the simple widget application, close the *Simple GUI* window.

Distributing IDL Programs

Once an application has been written in IDL, the user may wish to distribute this program to friends, colleagues, or their own customers. IDL programs can be stored and distributed as source code in ASCII text files with a "*.pro*" extension, or they can be compiled within a development copy of IDL and saved to runtime binary files with a "*.sav*" extension. Creating a runtime binary save file for an application allows the programmer to create a distributable version of their application that does not contain the original source code, thereby protecting the intellectual property rights of the developer. Furthermore, the runtime binary save file version of a program can be executed in the runtime environment of IDL, whereas a source code file requires that the end-user have a full development copy of the IDL software package in order to compile the program before execution. Creation of a runtime binary save file version of a program involves using either the *Projects* built into the IDL Development Environment, or the *.COMPILE*, *RESOLVE_ALL*, and *SAVE* routines built into the IDL language library.

IDL has an architectural (and distribution) paradigm that is very analogous to Java since they are both interpreted computer languages. There are basically 3 different varieties of the IDL software package that can be used to execute programs written in the IDL language :

- IDL (full developer's copy)
- IDL Runtime
- IDL Virtual Machine

The primary difference between these varieties of IDL is that the full developer's copy allows the user to create, edit, modify, compile, execute, and save IDL programs. In contrast, both the IDL Runtime and IDL Virtual Machine versions of IDL can only be used to execute pre-compiled runtime binary versions (**.sav* file) of a program. Consequently, users of the IDL Runtime or the IDL Virtual Machine cannot modify the IDL programs that are being executed.

In the following exercise, the "*muscle_view*" program that was created earlier will be compiled and saved out to a runtime binary file on disk that can be executed in either IDL Runtime or the IDL Virtual Machine. This can be accomplished with the *SAVE* procedure, which is used to save either data variables or the compiled routines within the current software session to IDL's proprietary binary save file format. The IDL save file format is encrypted (and is not documented) so it is impossible to reverse-engineer a program stored in a "**.sav*" file and obtain the original source code. Calling the *SAVE* procedure with the *ROUTINES* keyword set will save all currently compiled user-defined procedures and functions within the current IDL session. Consequently, it is a good idea to reset the IDL session and start fresh so that none of the other programs that have been compiled thus far are included in the save file that is created :

1. IDL> `.reset_session`

Next, compile the "*muscle_view*" program that was created in the earlier exercise by utilizing the *.COMPILE* executive command :

2. IDL> `.COMPILE muscle_view`
   ```
   % Compiled module: MUSCLE_VIEW.
   ```

The Runtime and Virtual Machine versions of the software are basically the IDL interpreter provided in a series of library files. This interpreter includes all of the internal system routines within the IDL language. However, it does not include the routines within the IDL library that are written in IDL itself and are distributed as source code **.pro* files in the "*lib*" subfolder of a developer's copy installation. For example, the *FILEPATH* function from the IDL library that this "*muscle_view*" program utilizes is actually distribute with the software in the following source code file :

- **Windows:** *C:\RSI\IDL##\lib\filepath.pro*
- **UNIX et al.:** */usr/local/rsi/idl_#.#/lib/filepath.pro*
- **Mac OS X:** */Applications/rsi/idl_#.#/lib/filepath.pro*

When the "*muscle_view*" program is executed within a full developer's copy of IDL, the "*filepath.pro*" source code file is automatically located and compiled on-the-fly because the "*lib*" subfolder of the installation is part of the default path. Since the Runtime and Virtual Machine versions of IDL cannot compile programs from their ASCII text source code, these routines cannot be distributed in their original **.pro* file form. Consequently, the programmer must be sure to resolve and compile all of the external IDL routines that their program utilizes that are not internal system routines built into the interpreter libraries. Fortunately, IDL contains a convenient procedure that can be used by the IDL programmer in order to resolve all of these dependencies named *RESOLVE_ALL* :

3. IDL> `RESOLVE_ALL`
   ```
   % Compiled module: RESOLVE_ALL.
   % Compiled module: LOADCT.
   % Compiled module: FILEPATH.
   % Compiled module: PATH_SEP.
   % Compiled module: UNIQ.
   ```

Notice that the *RESOLVE_ALL* procedure will locate and compile not only the *LOADCT* and *FILEPATH* routines from the IDL library that this program explicitly calls, but will also resolve any subsequent routines that these programs happen to call that are not already compiled (in this case, *PATH_SEP* and *UNIQ*). Under initial inspection, it may not be obvious to the programmer that this "*muscle_view*" program relies on the *PATH_SEP* and *UNIQ* routines from the external IDL library because they do not explicitly appear within the program, and this is the benefit of utilizing the *RESOLVE_ALL* procedure when creating distributable applications.

Finally, the *SAVE* procedure can be called with the *ROUTINES* keyword set in order to save all of the currently compile procedures and functions to an IDL runtime binary save file. Use the *FILE* keyword to specify the appropriate output filename, which is the name of the primary program module followed by a *.sav* extension :

4. IDL> `SAVE, /ROUTINES, FILE='muscle_view.sav'`

This operation will create a new file named "*muscle_view.sav*" within the current working directory.

5. Navigate to this folder and attempt to locate the file named "*muscle_view.sav*". If this file is not found within the folder, then execute the following IDL statements in order to determine the current working directory and locate the "*muscle_view.sav*" file :

```
IDL> CD, CURRENT=current
IDL> PRINT, current
```

6. Once the "*muscle_view.sav*" file has been located, it can be executed within IDL Runtime using the appropriate execution methodology based on operating system :

- **Windows:** Simply double-click on the "*muscle_view.sav*" file.
- **UNIX, Linux, & Mac OS X:** At a shell or X11 terminal prompt, navigate to the folder that contains the "*muscle_view.sav*" file and execute the following command :

```
idl -rt=muscle_view.sav
```

Once this is accomplished a new IDL process will launch, the "*muscle_view*" program will be executed, and the IDL process will shut-down.

7. Finally, the same program can be executed within the IDL Virtual Machine by performing the following steps :

- **Windows:** Select "*Start > Programs > RSI IDL #.# > IDL Virtual Machine*".
- **UNIX, Linux, & Mac OS X:** At a shell or X11 terminal prompt, navigate to the folder that contains the "*muscle_view.sav*" file and execute the following command :

```
idl -vm=muscle_view.sav
```

8. The IDL Virtual Machine splash screen will appear, and the user must click on this splash screen in order to continue.
9. On Windows, the user will be prompted with a dialog to select the "*muscle_view.sav*" program file on the harddrive.

Once this is accomplished a new IDL process will launch, the "*muscle_view*" program will be executed, and the IDL process will shut-down.

A discussion on the creation of distributable applications, utilization of the Projects built into the IDL Development Environment, and the exact differences between IDL Runtime and IDL Virtual Machine is beyond the scope of this tutorial. For more information please consult the *Building IDL Applications* documentation manual.

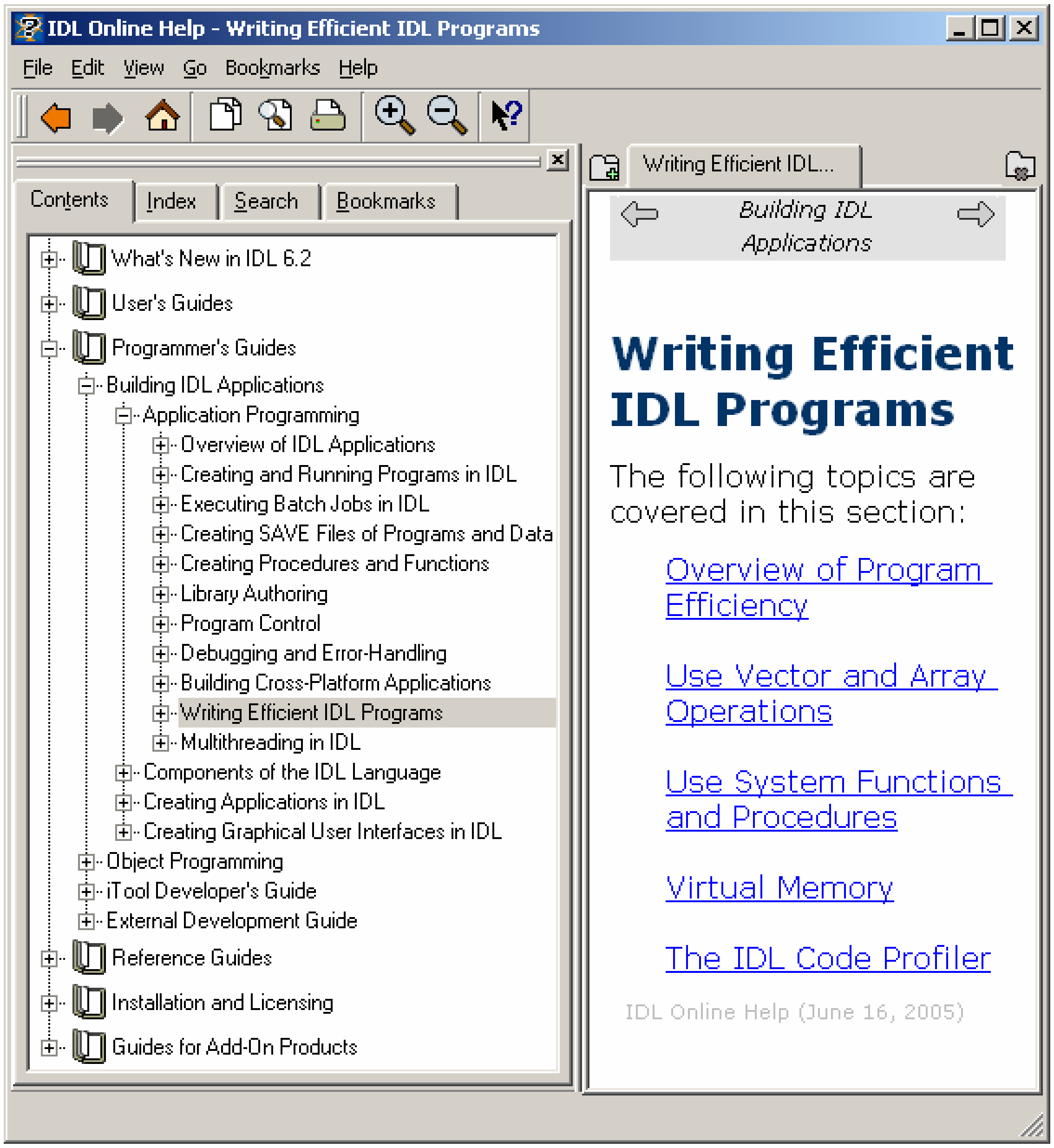
Writing Efficient IDL Programs

Knowledge of IDL's internal design and implementation, along with careful memory management, can be exploited 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 compiled programming 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 vector and array operations rather than loops wherever possible.
- Try to avoid loops with high repetition counts.
- Use IDL system functions and procedures wherever possible.
- Access array data in machine address order (IDL is row-major).
- Pay attention to expression evaluation order.
- Avoid *IF … THEN … ELSE* code block statements if possible, especially within loops.
- Use only the highest precision (variable data types) necessary during computations.
- Eliminate invariant expressions.
- Make use of the *TEMPORARY* function.
- Utilize the IDL Code Profiler.

Attention also must be given to algorithm complexity and efficiency, as this is usually the greatest determinant of resources used. For a more detailed discussion on the subject of writing efficient IDL programs (along with some code examples), please consult the *Building IDL Applications* documentation manual included with the IDL online help system [Fig. A-1].


IDL Online Help - Writing Efficient IDL Programs

File Edit View Go Bookmarks Help
Contents Index Search Bookmarks
Writing Efficient IDL...

- What's New in IDL 6.2
- User's Guides
- Programmer's Guides
  - Building IDL Applications
    - Application Programming
      - Overview of IDL Applications
      - Creating and Running Programs in IDL
      - Executing Batch Jobs in IDL
      - Creating SAVE Files of Programs and Data
      - Creating Procedures and Functions
      - Library Authoring
      - Program Control
      - Debugging and Error-Handling
      - Building Cross-Platform Applications
      - Writing Efficient IDL Programs
      - Multithreading in IDL
    - Components of the IDL Language
    - Creating Applications in IDL
    - Creating Graphical User Interfaces in IDL
  - Object Programming
  - iTool Developer's Guide
  - External Development Guide
- Reference Guides
- Installation and Licensing
- Guides for Add-On Products

Building IDL Applications

Writing Efficient IDL Programs

The following topics are covered in this section:

Overview of Program Efficiency

Use Vector and Array Operations

Use System Functions and Procedures

Virtual Memory

The IDL Code Profiler

IDL Online Help (June 16, 2005)

Figure A-1: The discussion on Writing Efficient IDL Programs